Méthodes et outils pour la programmation des systèmes cyber-physiques

Louis Viard

UNIVERSITÉ DE LORRAINE

École doctorale IAEM Lorraine

Méthodes et outils pour la programmation des systèmes cyber-physiques

THÈSE

présentée et soutenue publiquement le 1er avril 2021

pour l'obtention du

Doctorat de l'Université de Lorraine

(mention informatique)

par

Louis Viard

Composition du jury

Président :	Dominique Mery	
Rapporteurs :	Yamine Aït-Ameur	Professeur, ENSEEIHT/IRIT, Toulouse
	Liliana Cucu-Grosjean	Chargé de Recherche, Inria, Paris
Examinateurs :	François Despaux	Ingénieur de Recherche, Alérion, Nancy
	Dominique Mery	Professeur, Université de Lorraine, Nancy
Directeurs de thèse :	Laurent Ciarletta	Maître de Conférences, Université de Lorraine, Nancy
	Pierre-Etienne Moreau	Professeur, Université de Lorraine, Nancy

Laboratoire Lorrain de Recherche en Informatique et ses Applications — UMR 7503

Mis en page avec la classe thesul.

La plus subtile de toutes les finesses est de savoir bien feindre de tomber dans les pièges que l'on nous tend, et on n'est jamais si aisément trompé que quand on songe à tromper les autres.

– Maximes et Réflexions morales,
La Rochefoucauld

Table des matières

Table des figures

Avant-propos

Le présent ouvrage rapporte trois années de recherche en tant que doctorant au sein du Loria (Laboratoire lorrain de recherche en informatique et ses applications), entre les équipes SIMBIOT, s'intéressant aux systèmes cyber-physiques « intelligents », et MOSEL, étudiant les méthodes formelles et leurs applications. La contribution résultante s'inscrit de fait à l'interface de ces domaines, eux-mêmes déjà riches et variés. L'union de ces thématiques sourd d'interrogations sur la sûreté de fonctionnement des systèmes cyber-physiques, d'une nécessité accrue de maîtrise des risques qu'ils produisent. La modernité de ces questions procède tout d'abord de la jeunesse de leurs objets d'étude : les systèmes cyber-physiques sont les fers de lance d'une révolution que préparent les pays développés, l'avènement d'une industrie 4.0 consacrée par le plan stratégique allemand éponyme ou son équivalent *Made-in-China 2025* dans l'empire du Milieu. Après les révolutions du charbon, de l'électricité, et de l'électronique, nous voici promis à la convergence du numérique et du physique, à la conjugaison du virtuel et du réel. Les acteurs ayant cette double nationalité sont les systèmes cyber-physiques, dont beaucoup sont encore en gestation. Dans la vaste quête de l'intelligence, ceux-ci se démarquent de leurs prédecesseurs notamment par une autonomie accrue ainsi que leur évasion : le robot industriel confiné dans sa cage laisse la place au cobot, un robot collaboratif évoluant sans danger au milieu des humains. De là, l'usine intelligente met en réseau les humains, les informations, les ressources, et les systèmes cyber-physiques, amenant à une production distribuée sur différents sites et personnalisable : la fabrication individuelle se substitue à la production de masse.

Le second motif pour s'intéresser aujourd'hui à la gestion de risques des systèmes cyber-physiques est le risque lui-même. Celui-ci n'est pourtant pas un concept nouveau : on le retrouve déjà associé aux assurances pour socialiser les pertes en marchandises maritimes qu'encouraient les armateurs italiens au XIVème siècle. Le développement théorique du risque doit alors à Blaise Pascal qui fournit un cadre pour en calculer l'aléa, et à l'évolution à travers les âges de la perception des catastrophes, celles-ci étant successivement le fruit d'une damnation, d'une faute puis d'un accident. L'une des caractéristiques du risque est en effet l'absence de faute imputable, l'accident est accepté inéluctable et, à la recherche d'un responsable la société substitue une souscription assurantielle. Cette vision providentielle de l'assurance domine depuis le XXème siècle, mais ne marque pas la fin de l'évolution de la perception du risque. En substance, elle semble cantonner le risque à un problème d'actuaires, c'est-à-dire de calcul de primes d'assurance à partir de séries statistiques. Alors que les secteurs couverts par des assurances se multiplient, leur extension à des risques construits sur des dangers qui ne se laissent aisément observés ou à effet différé ébranle ce *statu quo*. Le citoyen cherche à prendre part à l'analyse du risque, l'expert est décontenancé par ses prises de position qui ne s'expliquent guère simplement à partir des aléas [1] et des vulnérabilités [2], et la sociologie se saisit également du sujet, de manière

1. « probabilité qu'un phénomène accidentel produise en un point donné des effets d'une intensité donnée au cours d'une période déterminée. » source : INERIS

2. « appréciation du rapport entre les effets d'un danger auquel est exposé une cible et les dommages qu'elle

constructiviste (*e.g. Risikogesellschaft* de Beck, pour une introduction voir [190]), ou positiviste (*e.g.* les cindyniques [122]). Ces développements sont contemporains des accidents de Seveso et Tchernobyl, de la maladie de la vache folle ou encore de l'affaire du sang contaminé. Si la quantification du risque conserve sa forme de produit entre l'aléa et la vulnérabilité, son analyse se diversifie. Chevassus-au-Louis [67] le regarde par exemple sous six angles pour assister l'acte de décision

- trois variables aggravantes : sévérité, acceptabilité, plausibilité
- trois variables atténuantes : réductibilité, observabilité, réversibilité

Il convient d'intégrer au sein de l'acceptabilité l'appréhension qu'en a le profane et non uniquement une valeur numérique arbitraire (*e.g.* un mort supplémentaire par million d'habitants). En somme, le risque est une construction humaine sur le danger, et son traitement ne cesse de se complexifier. N'envisager sa gestion que sur l'axe unidimensionnel de l'expert mène au-devant de difficultés dans son acceptation sociale. En l'occurence, la prolifération des systèmes cyber-physiques au vu et au su de tous ne manquera pas d'inquiéter et de mobiliser le profane ; au demeurant, elle suscite d'ors et déjà bien des fantasmes.

La singularisation du produit qu'ambitionne l'industrie 4.0 nécessite de repenser localement la gestion des risques, en particulier quand celui-là est un système cyber-physique. La difficulté d'une certification d'un produit croît fortement lorsque l'on ne connaît plus sa finalité. Or, la modification par opération d'une plateforme drone dans une logique « ad-hocquiste » ne permet plus de présager de son utilisation, et l'on serait bien en peine de donner indifféremment des garanties sur son emploi dans le cadre d'une inspection d'infrastructure portuaire ou de livraison en montagne. En conséquence, la législation évolue vers une prise en charge, au moins partielle, de l'analyse de risque par l'opérateur. Demeure la question des outils et méthodes dont disposera cet utilisateur pour opérer ces systèmes cyber-physiques : il lui faudra non seulement veiller à la sûreté de fonctionnement, mais également communiquer à son sujet pour convaincre directement les autorités régulatrices et indirectement les citoyens.

Le défi que nous avons esquissé, par sa nature titanesque, n'est pas à la mesure d'un travail de thèse. Il embrasse en effet de trop nombreux champs de compétences, de la conception de la plateforme matérielle à son opération dans son cadre applicatif en passant par son outillage logiciel, sa gestion des risques, et l'étude sociologique préalable à son utilisation. Nos contributions que nous présenterons dans la seconde partie de cet ouvrage participent de l'environnement logiciel de ces systèmes. Si l'ambition qui présida à notre projet fut le développement d'outils et de méthodes pour la programmation de systèmes cyber-physiques mobiles, il est préjudiciable d'abandonner cette vision systémique. Notre approche est tout d'abord créative : nous définissons un langage dédié à la programmation des systèmes cyber-physiques invitant à protocoliser la conduite à tenir en cas d'écart significatif à l'exécution nominale de la mission, par exemple en cas de concrétisation d'un aléa. Nous en explicitons une notion de régularité, permettant de discriminer les missions selon la plausibilité de leur succès. Puis notre démarche est fondamentalement intégrative : nous adjoignons à la définition de programmes de mission des outils assistant leur conception, leur évaluation, et leur exécution. Le travail rapporté présentait également une dimension appliquée dans le cadre du projet `ceos`, portant sur l'inspection d'ouvrages par solution drone pour opérateurs d'importance vitale. Dès lors, les aéronefs télé-pilotés constituent naturellement les systèmes de prédilection qui sous-tendent les réflexions et développements que nous exposerons.

Le chapitre 1 présentera ce secteur : les prévisions sur l'usage civil des drones laissent entrevoir un domaine en pleine expansion, d'une grande complexité du fait de la kyrielle d'applications

subit » source : INERIS

existantes et envisagées, mais dont l'essor présuppose le traitement de verrous législatifs et techniques. Ce premier chapitre devrait ainsi offrir un recul suffisant pour comprendre le cadre dans lequel notre contribution s'intègre et les défis qui la motivent.

Si les drones sont les principaux cas d'usage employés dans les pages qui suivent, ils ne recouvrent qu'une partie des systèmes auxquels s'appliquent les outils et méthodes que nous présenterons. Ces systèmes, nous l'avons évoqué, sont dits « cyber-physiques » sans que cette classe ne possède de définition faisant l'unanimité. Proposer une définition des systèmes cyber-physiques dépasse le cadre de ce travail, cependant se contenter de cette indétermination sur nos objets d'étude ne saurait être une solution acceptable. Pour sortir de cette aporie, nous chercherons de quel signe[3] ils participent dans le chapitre 2, avant de nous intéresser plus particulièrement à la définition de mission pour systèmes cyber-physiques mobiles, qui constitue le domaine dans lequel s'inscrit notre contribution.

Les deux premiers chapitres montreront que les systèmes cyber-physiques mobiles sont, entre autres, des systèmes critiques. L'usage de drones, en zones peuplées par exemple, nécessite de donner des gages forts de sûreté de fonctionnement préalablement à l'obtention des autorisations afférentes. Le chapitre 3 donnera un aperçu d'approches essentiellement formelles qui s'intéressent à la vérification de systèmes cyber-physiques. En liminaire nous développerons une brève réflexion sur la notion de modèle. Les modèles sont en effet ubiquitaires en science, tant ils sont commodes, *viz.* manipulables, prouvables, économiques. Cependant ces qualités marquent leur différence avec les systèmes qu'ils modélisent, et la criticité susmentionnée des systèmes qui nous intéressent commande de garder cette disparité à l'esprit.

La deuxième partie de cet ouvrage sera consacrée à notre contribution. Celle-ci prend la forme d'un langage dédié aux systèmes cyber-physiques pour la définition de mission, `Sophrosyne`, entouré d'un ensemble d'outils. Le chapitre 4 donnera un aperçu général du langage, il en présentera les concepts généraux qui le caractérise et sa syntaxe concrète. Après cette ébauche, nous donnerons une sémantique formelle à `Sophrosyne` dans le chapitre 5 ; cette définition, quelque peu fastidieuse, reste un préalable à toute ambition d'analyse de missions.

Avant de détailler ces analyses, le chapitre 6 donnera corps au langage. Pour cela trois bibliothèques, écrites en `Python`, seront présentées. La première, éponyme, comprend les différents outils de la compilation de programmes `Sophrosyne`, de l'analyse lexicale à la génération de code. Cette bibliothèque permet ainsi de disposer de différentes représentations de programmes `Sophrosyne`, desquelles procèdent notamment l'exécution des missions, dont la plateforme d'exécution est assurée par la seconde bibliothèque `pysophrosyne`, et l'analyse formelle de leur correction, par l'intermédiaire de la troisième bibliothèque `symsophrosyne`.

Nous nous poserons alors la question légitime de la correction d'une mission. Pour cela, nous commencerons par présenter dans le chapitre 7 la logique dynamique différentielle que nous utiliserons comme formalisme pour le raisonnement logique. Le chapitre 8 définira alors dans cette logique la notion de correction d'une mission, en exhibant les propriétés qu'elle doit satisfaire. Ces propriétés se rangent dans deux grandes catégories usuelles : la vivacité et la sûreté.

Finalement, le chapitre 9 présentera une interface homme-machine de planification de mission que nous avons développée autour de `Sophrosyne`, intégrant les différentes briques que nous aurons présentées dans les chapitres précédents. En parallèle, nous l'illustrerons par des réalisations effectuées pour le projet `ceos`, dans lequel ces éléments ont été mis en oeuvre pour assurer une navigation automatique conforme à des contraintes de confinement géographique au-dessus d'infrastructures à inspecter.

3. Le signe est pris ici dans son sens saussurien, c'est-à-dire comme une correspondance d'un signifiant et d'un signifié [77]

Première partie

Cadre d'étude et état de l'art

Chapitre 1

Les drones : histoire et perspectives

Que Minos étende son empire sur la terre et sur les flots, le ciel du moins n'est pas sous ses lois.

Les Métamorphoses, Ovide

Les drones se parent aujourd'hui des atours d'une nouvelle conquête du ciel. Longuement associés à l'univers militaire, ils côtoient à présent le grand public, et les applications commerciales, encore balbutiantes, fleurissent. Bien qu'en bonne voie, leur intégration dans la société présuppose la levée de divers verrous, en particulier technologiques, réglementaires, et psychologiques. La complexité naturelle de ces systèmes – évolution dans un espace tridimensionnel souvent incertain voire inconnu, absence de solution simple, rapide et sûre d'interruption de vol – les érige en archétypes pour l'étude des systèmes cyber-physiques mobiles. De même, la difficile acceptation des drones, tant par les autorités régulatrices que par les citoyens inquiets face à cette immixtion dans leur quotidien, est symptomatique des oppositions que rencontrent les véhicules autonomes. Ce chapitre donnera un aperçu de ce secteur, et pourra illustrer des propos plus abstraits qui seront tenus dans la suite de cet ouvrage.

1.1 Éléments biographiques du drone

L'aventure du drone civile est récente, d'une quinzaine d'années bien qu'on en trouve des préfigurations dès les années 80 à l'instar des hélicoptères agricoles radiocommandés. Elle se démarque par la rapidité de son développement, sans doute facilité par un siècle d'expérimentations militaires. Il nous semble utile de rappeler quelques dates et événements importants dans cette histoire.

1.1.1 Le militaire

Les premiers drones voient le jour à l'issue de la Première Guerre Mondiale, notamment aux Etats-Unis, avec Elmer Sperry, co-inventeur du gyrocompas (avec Hermann Anschütz-Kaempfe), et l'ingénieur Peter Hewitt, qui firent le premier vol du *Hewitt-Sperry Automatic Plane* (figure 1.1) en septembre 1917, soit une vingtaine d'année après les débuts de l'aviation. De même, en France, le capitaine Max Boucher travaille à la réalisation d'un « avion automatique », qui se concrétise par un vol complet automatique d'un Voisin BN 3 en 1923. L'essor du drone militaire date toutefois de la Guerre Froide, et de son utilisation par les Etats-Unis comme moyen de

surveillance au Viet Nam et pendant la guerre du Kippour. Aujourd'hui, Israël est l'un des principaux acteurs de ce secteur, avec les Etats-Unis et la Chine. Ils bénéficient d'une situation avantageuse pour le développement de cette technologie : l'instabilité des frontières fournit un terrain d'expérimentation privilégié, réduisant les cycles de développement. Ronen Nadir, ancien commandant de l'armée israélienne et fondateur de la société BlueBird Aero Systems qui vend des drones de combat expliquait ainsi que seulement quatre mois s'étaient écoulés entre l'idée initiale et la présentation d'un drone à un client [4]. De nombreux industriels israéliens du secteur drone, y compris civil, sont d'anciens militaires, ce qui favorise le développement des secteurs commerciaux.

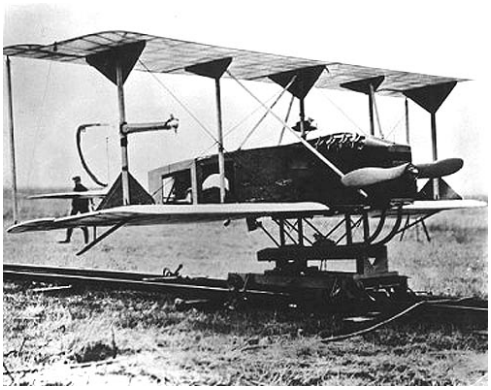


FIGURE 1.1 – Le *Hewitt-Sperry Automatic Airplane*

La France souffre d'un retard dans sa filière du drone militaire, ainsi que dans l'équipement de ses armées, bien qu'une politique d'accélération des acquisitions est à présent en place [73]. Les États-Unis conservent une position hégémonique avec leur flotte nombreuse et variée, et largement utilisée. L'armée s'est modernisée pour faire face aux contraintes des guerres post 11 septembre (opinion publique, théâtres variées, rapidité de déploiement), et l'industrie, déjà structurée, exerce son influence sur les politiques publiques et sur la constitution de la réglementation [109]. Israël s'est imposé toutefois comme producteur avec un écosystème innovant et une politique de soutient constante. La Chine émerge, en produisant des drones réputés moins performants mais bien moins coûteux. Elle équipe notamment des pays que d'autres constructeurs fournissent plus difficilement : Irak, Pakistan, Arabie Saoudite, Égypte, *etc.* La Russie est en retard, et en Europe, le Royaume-Uni reste le mieux équipé. Des initiatives existent toutefois, comme en Serbie, où l'industrie aéronautique militaire s'oriente sur le drone afin de renouveler le parc aéronautique hérité de la Yougoslavie [163]. Plusieurs drones sont déjà développés localement, essentiellement pour l'observation et la reconnaissance. Des transferts de technologie ont également été initiés avec la Chine, la Serbie ayant fait l'acquisition de drones de combats Wing Loong II.

La production militaire se diversifie, le nombre de drones capables de tirer des missiles a doublé ces cinq dernières années [73]. Des organisations non-étatiques, dont terroristes, se sont aussi équipées, souvent à partir de solutions commerciales utilisées pour la surveillance ou simplement modifiées pour le combat. La première trace d'une relation entre drones et terrorisme date des années 90, la secte Aum Shinrikyo songeait alors à utiliser un hélicoptère télé-piloté pour disperser du gaz sarin [162]. Toutefois, jusque récemment, les organisations terroristes employaient les

4. « "You will not believe it, but it took only four and a half months" from the first concept, scribbled on paper, "until this UAV was demonstrated to the first customer," Nadir said. » [34]

drones essentiellement pour de la surveillance ou pour des communications stratégiques. L'Etat Islamique en a ainsi fait longuement usage, tout d'abord pour de la connaissance situationnelle de théâtre, de la reconnaissance, de la surveillance et de l'acquisition d'images et vidéos pour la propagande, puis, à partir de 2016, en les équipant d'engins explosifs improvisés [36]. L'armée de terre utilise encore le système de drone tactique intérimaire (SDTi), dont les limites en terme d'autonomie, de qualité, et d'évolution en font un système peu séduisant. Son utilisation intensive a grevé le budget initialement prévu. L'armée de l'air emploi des drones MALE israéliens, solution initialement temporaire en l'attente de MALE européens qui n'ont finalement jamais vu le jour.

L'armement des drones suscite encore de nombreux débats, en Europe, mais aussi dans les pays les employant déjà (Etats-Unis, Israël). Ces équipements nécessitent une autonomie de décision, notamment un pilotage automatique en environnement non permissif, dans lequel les communications et la discrétion ne peuvent être garanties. Les projets de coopération sur le plan européen ont successivement échoué : le *Watchkeeper* ainsi que le *Telemos* avec les britanniques, le *Talarion* avec les gouvernements allemand, et espagnol [104]. En conséquence, les drones MALE étrangers (américains ou israéliens) sont ubiquitaires dans les armées européennes. Une nouvelle coopération a toutefois été amorcée avec l'Allemagne, l'Espagne et l'Italie et leurs industriels. Toutefois, des différences de besoins opérationnels alourdissent la tâche.

1.1.2 Le civil

Afin de donner un aperçu historique, nous présentons tout d'abord une sélection chronologique d'événements qui façonnent le drone civil depuis une dizaine d'années. En 2007, Chris Anderson crée le forum `DIYdrones.com`, dont la communauté donne naissance à l'autopilote ArduPilot [4]. Après une décennie le forum compte plus de 90000 membres. Dès 2007, la police française s'est équipée de drones Elsa afin d'assurer une surveillance aérienne de manifestations ou de violences urbaines [35]. Le premier drone "grand public" est dévoilé en 2010 au CES Las Vegas, il s'agit de l'AR.Drone de Parrot, à moins de 300€ pièce [143]. En 2011, Chris Anderson fonde 3Drobotics, l'année d'après Franck Wang crée Da Jiang Innovation (DJI). En 2012, le ministère des Transports signe les deux premiers arrêtés réglementant l'utilisation de drones civiles [135]. Ceux-ci sont extrêmement restrictifs, le vol n'étant possible qu'à vue, hors de zone interdites (tels que les aérodromes et les centrales nucléaires), avec une limite d'altitude d'au plus 150 mètres, et sans risque manifeste de dommage aux tiers au sol. En 2013 un drone franchit le Bosphore (soit une distance d'environ 1 kilomètre) [99] ; Amazon annonce par ailleurs Prime Air, un programme de livraison par drone [2]. Afin d'alerter l'opinion sur le risque sécuritaire posé par les drones, le parti pirate fait voler un drone au-dessus de la tribune de la chancelière allemande en campagne électorale [1]. En 2014, Google rejoint la course avec Project Wing [17], projet de livraison de bien de consommation (nourriture et médicaments) dans le Queensland en Australie. En 2014 et 2015, la France voit les deux tiers de ses centrales nucléaires survolées par des drones, ainsi que d'autres sites sensibles (palais de l'Elysée, tour Eiffel, Invalides, ambassade américaine) [160]. L'ONG Greenpeace fera d'ailleurs s'écraser en 2018 un drone en forme de superman sur la centrale nucléaire du Bugey, près de Lyon, afin de dénoncer la vulnérabilité des sites nucléaires [26]. Le Conseil pour les Drones Civils est créé en 2015 afin d'accompagner le rapide développement du secteur en France et de coordonner les efforts de ses différents acteurs. Depuis 2016 et la présentation de l'Ehang 184 au CES Las Vegas [50], les projets de transport par aéronef autonome fleurissent, à l'instar du Lilium Jet, un taxi aérien qui embarquait en 2017 deux passagers dans son vol d'essai près de Munich et qui prévoit 5 places pour 2025 [11], ou de l'entreprise allemande Volocopter, qui a fait des premiers tests à Dubai en 2017 [23]. Après la société Zipline au Rwanda

(2016) [25], les drones Matternet livrent également des médicaments et des poches de sang, notamment en Afrique, mais aussi en Suisse [28] ou en Floride [71]. Le président vénézuelien est la cible d'une attaque de deux drones DJI Matrice 600 qui portaient 1 kg d'explosifs chacun [74]. L'attentat échoue mais blesse sept personnes.

L'opinion publique occidentale était initialement réticente aux drones, qui restaient inéluctablement associés au militaire et à des emplois controversés. Un changement de mentalité a été depuis progressivement mené, en communicant largement sur des emplois à des fins sociétales ou humanitaires, tels que le transport de poches de sang, la surveillance environnementale, ou l'assistance aux services de secours et d'intervention en gestion de crise [139]. En Europe, le vide législatif a conduit chaque pays a faire ses propres lois, ce qui a parfois envoyé des signaux contraires. La Suède par exemple, a interdit l'usage de drone embarquant une caméra par un arrêt de sa Cour administrative suprême en 2016 sauf si l'utilisateur a obtenu un permis spécial.

Avant que le marché du drone commercial n'explose, plusieurs sujets restent encore à traiter. La société a encore des inquiétudes quant à son utilisation, en particulier concernant la sûreté et la sécurité, le respect de la vie privée, et la responsabilité [161]. La législation européenne survole la question sécuritaire, et les solutions de l'aviation traditionnelle, reposant par exemple sur le contrôle en aéroport, ne sont pas toujours adaptées à l'usage sauvage de drones [116]. La navigation, généralement basée sur la géo-localisation, est connue pour être vulnérable à des attaques d'usurpation de GPS, permettant de détourner le drone [184]. Les drones embarquant des caméras, les enregistrements vidéos des vols interrogent quant au respect de la vie privée, tant pour les personnes filmées dans l'espace publique que pour les nouveaux angles de vues qu'ils impliquent sur les propriétés privées. Des débats nationaux seront nécessaires pour identifier des nouvelles règles relatives à la vie privée, ou réaffirmer celles préexistantes. Ces considérations constituent de nouveaux problèmes pour les agences de régulation de l'aviation civile qui n'y avaient jusqu'alors que rarement été confrontées. La maturité sur les questions de responsabilité se traduira par le développement des assurances, qui doivent répondre en premiers lieux aux risques de dommage à autrui et de manquements au respect de la vie privée. Elles font toutefois face à des difficultés similaires à celles des autorités délivrant les autorisations de vol : le secteur a été porté par des amateurs de culture *maker*, d'où procède une kyrielle d'engins modulaires, à composants *ad hoc* parfois rudimentaires, et aux logiciels *open source* pour lesquels le suivi des changements est souvent inapplicable.

1.2 Le marché du drone civil

La variété des utilisations de drones et la dynamique de développement transparaît dans la répartition des acteurs selon les secteurs du marché (voir figure 1.2). Depuis quelques années, les activités se diversifient et des entreprises se redéfinissent, à l'instar de 3DR qui est progressivement passé du drone de loisir à la construction de solutions commerciales et d'équipements, en particulier pour le gouvernement fédéral des États-Unis [32]. Le segment de la construction de drone reste dominé par les États-Unis et la Chine (cf figure 1.3), et plusieurs acteurs historiques ont réussi à s'inscrire durablement dans le paysage, tels que DJI, Parrot, ou 3DR. Les prévisions actuelles font entrevoir un secteur tiré par les pays développés, la plus forte croissance revenant au marché asiatique (voir figure 1.4). En Europe, les secteurs commerciaux ayant déjà amorcé leur transition devraient se confirmer comme les moteurs du marché, la livraison concentrant en particulier beaucoup d'attentes (figure 1.5). Les applications commerciales actuelles et futures du drone civil sont nombreuses et variées ; la section qui suit devrait en donner un aperçu. Pour une étude prospective plus détaillée, nous renvoyons vers [172, 157].

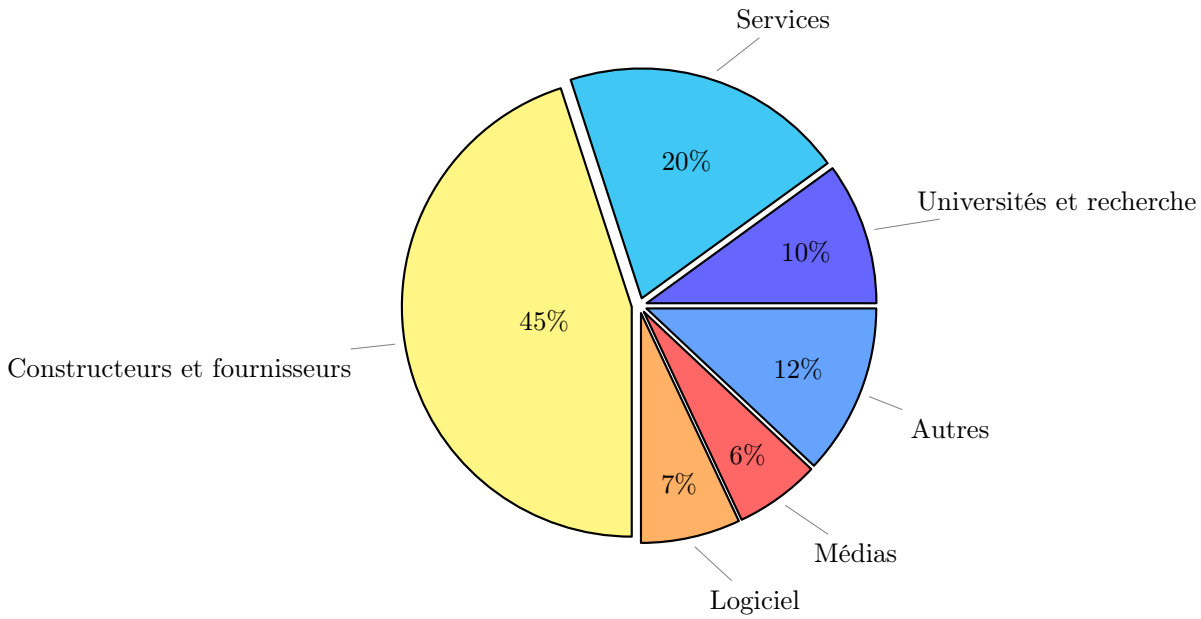
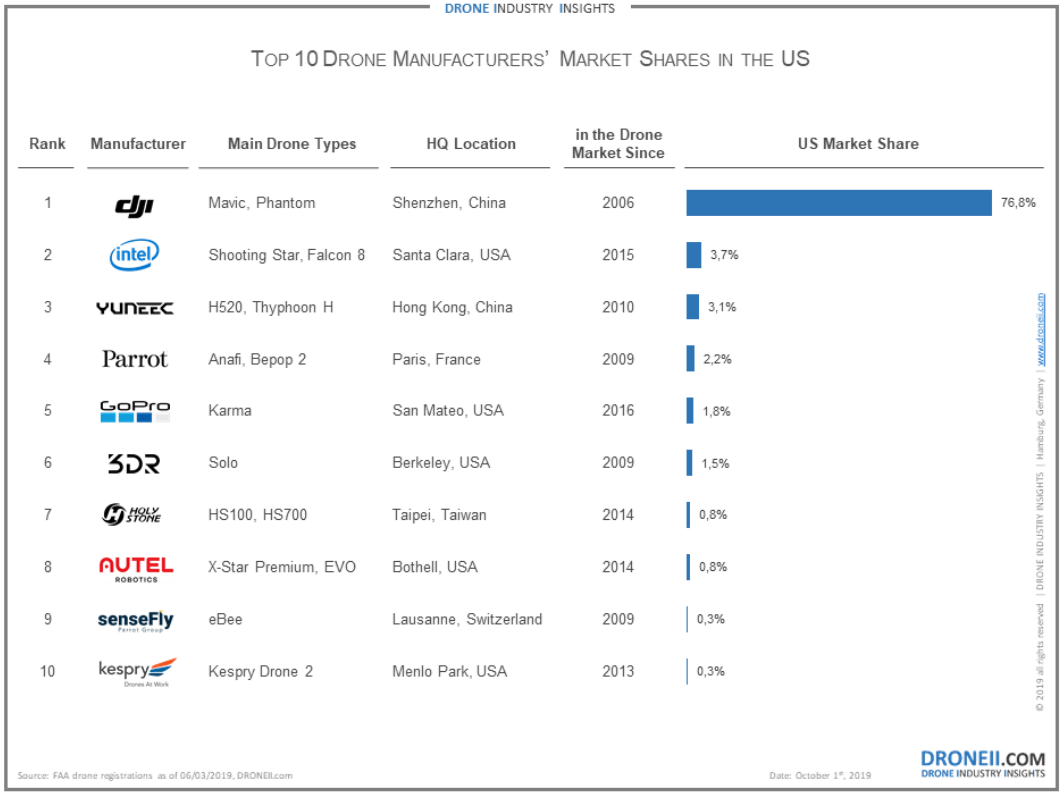


FIGURE 1.2 – Répartitions des principaux acteurs du marché drone en 2016, à partir de 711 entreprises de 49 pays d'après [94]


DRONE INDUSTRY INSIGHTS

TOP 10 DRONE MANUFACTURERS' MARKET SHARES IN THE US

Rank	Manufacturer	Main Drone Types	HQ Location	in the Drone Market Since	US Market Share
1	dji	Mavic, Phantom	Shenzhen, China	2006	76,8%
2	intel	Shooting Star, Falcon 8	Santa Clara, USA	2015	3,7%
3	YUNEEC	H520, Thyphoon H	Hong Kong, China	2010	3,1%
4	Parrot	Anafi, Bepop 2	Paris, France	2009	2,2%
5	GoPro	Karma	San Mateo, USA	2016	1,8%
6	3DR	Solo	Berkeley, USA	2009	1,5%
7	HOLY STONE	HS100, HS700	Taipei, Taiwan	2014	0,8%
8	AUTEL ROBOTICS	X-Star Premium, EVO	Bothell, USA	2014	0,8%
9	senseFly	eBee	Lausanne, Switzerland	2009	0,3%
10	kespry	Kespry Drone 2	Menlo Park, USA	2013	0,3%

Source: FAA drone registrations as of 06/03/2019, DRONEII.com

Date: October 1st, 2019

DRONEII.COM
DRONE INDUSTRY INSIGHTS

FIGURE 1.3 – Les principaux constructeurs de drones selon le marché des Etats-Unis en 2019, repris de [91]

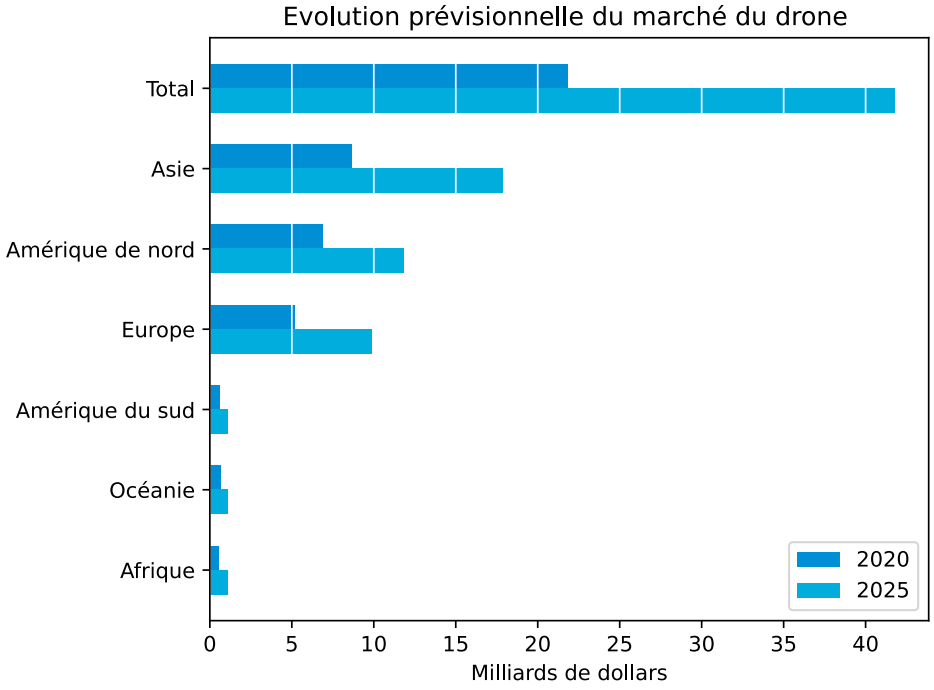


FIGURE 1.4 – Evolution prévisionnelle du marché mondial du drone par continent (produit à partir de données de [92])

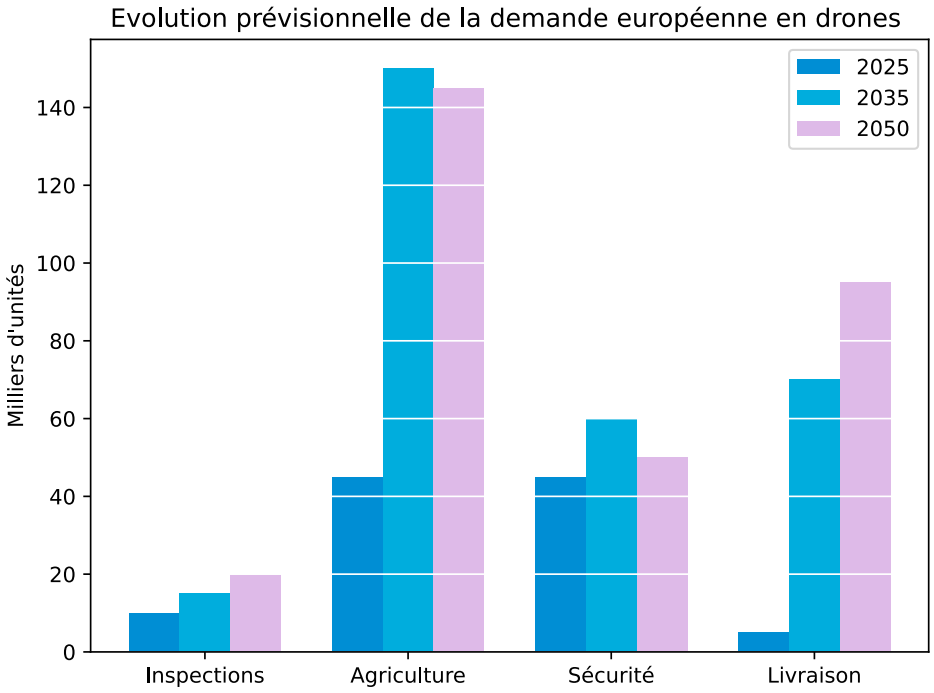


FIGURE 1.5 – Evolution prévisionnelle de la demande européenne en drones pour les principaux secteurs commerciaux (produit à partir de données de [172])

1.2.1 Le secteur de la gestion d'infrastructures

Un des axes principaux de développement des applications commerciales de drones concernent les infrastructures (voir figure 1.6) ; les drones ont un rôle à jouer de leur construction à leur maintenance. Leur facilité de déploiement et les nouveaux angles de vue qu'ils fournissent permettent un meilleur repérage du terrain avant construction et l'obtention de données pour le suivi d'avancement. Ils contribuent également à la vérification des aspects sécuritaires des sites de construction. Concernant la maintenance d'infrastructures, les drones complètent ou remplacent progressivement des inspections faites manuellement, qui sont coûteuses et nécessitent parfois l'accès à des zones dangereuses. Ces inspections s'appliquent en premier lieu aux infrastructures énergétiques ; on distingue alors des inspections statiques, pouvant s'effectuer avec un drone captif ou, à défaut, impliquant un vol à vue, des inspections de longue élongation à l'instar de la maintenance de pipeline ou de lignes électriques. Trois exemples d'inspection d'infrastructures seront présentés dans le chapitre C comme applications des travaux que nous détaillons dans cet ouvrage.

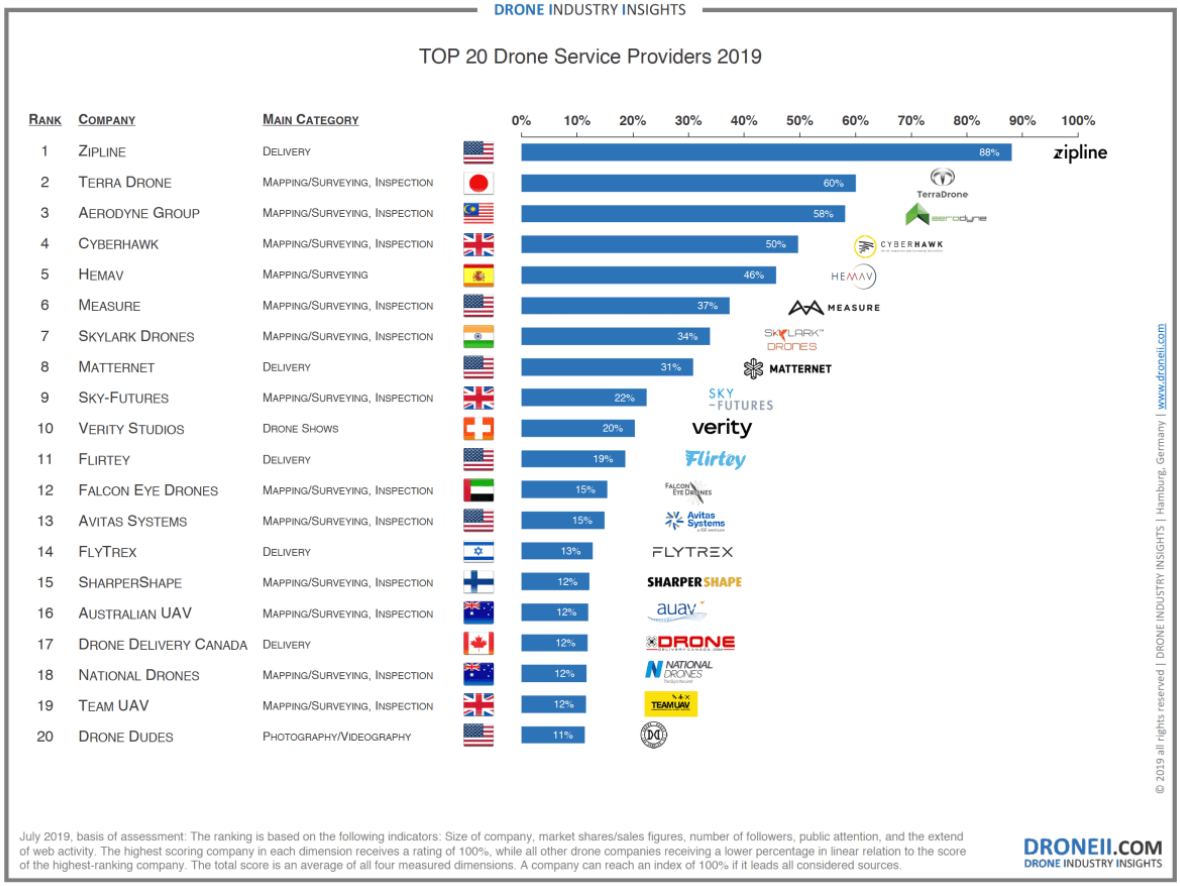


FIGURE 1.6 – Les 20 principaux fournisseurs de services drones en 2019, repris de [93]

1.2.2 Le secteur agricole

L'agriculture est un domaine d'application historique des drones. Des entreprises comme Yamaha Motor Co. au Japon se sont intéressés à ces applications depuis les années 90 [97]. Avec l'augmentation des capacités en charge utile, les drones sont notamment attendus pour la dispersion de semences et de produits chimiques. Le marché chinois, avec ses vastes étendues agricoles et un coût de la main d'oeuvre croissant est particulièrement demandeur. Par ailleurs, l'équipement en capteurs sophistiqués associés aux capacités de traitement de ces informations permet de diminuer l'utilisation de pesticides et d'augmenter les rendements. Pour nourrir une population mondiale croissante, l'amélioration des rendements demeure un axe principal de développement. Les exploitations agricoles sont souvent disposées à introduire de nouveaux outils technologiques pour améliorer leur production ; elles ont ainsi introduit l'imagerie satellitaire dans leur processus, afin de surveiller (*e.g.* détection de zones en stress hydrique, télé-épidémiosurveillance), ou de prédire (*e.g.* tonnage des récoltes) [126]. Les drones devraient se substituer à l'imagerie satellitaire pour plusieurs de ses usages dans l'agriculture de précision, les aéronefs étant plus flexibles et plus précis que des images sensibles à la couverture nuageuse et devant être commandées à l'avance.

1.2.3 Le secteur des médias et du divertissement

L'industrie du cinéma s'est rapidement emparée des drones. Ceux-ci permettent d'obtenir des plans qui auparavant nécessitaient de lourds investissements techniques et financiers. Les drones sont également plébiscités pour la réalisation de documentaires sur la faune et la flore sauvages ; ils diminuent les perturbations engendrées sur l'écosystème par la présence humaine ou par le souffle important produit par un hélicoptère, en plus d'éliminer des risques humains.

Les diffuseurs d'événements sportifs se montrent aussi intéressés par les possibilités qu'offrent les drones. Lors des Jeux Olympiques de Sochi, en 2014, les épreuves de slopestyle ont été filmées par un drone [148]. Depuis, cette usage tend à se développer, comme lors du Grand Prix de Formule 1, en France, en 2019 [24] ou dans le cadre de projets encore à l'état de recherche à l'instar de courses cyclistes [61]. Les drones fascinent lors de grand spectacles avec vol autonome en formation : la société Pixiel (depuis rachetée par Delta Drone) a collaboré avec le Puy du Fou pour produire une flotte capable d'effectuer des chorégraphies aériennes [156]. Plusieurs villes ont de même choisi d'effectuer des spectacles drones pour le nouvel an, parmi lesquelles Shanghai en 2020. Toutefois, la robustesse de ces événements est encore à améliorer, Bruges ayant par exemple annulé le sien pour cause de problèmes techniques et de brouillard [48].

1.2.4 Le secteur de la livraison et du transport

Les commerces, en particulier les e-commerces, sont toujours en quête d'une réduction du coût de livraison. Les drones sont pressentis pour améliorer le dernier kilomètre, que cela soit pour le coût ou la vitesse. Amazon et Google sont des acteurs de ce secteur, avec respectivement Amazon Prime Air [2] et le Project Wing [17]. Airbus expérimente sur un segment particulier, celui de la livraison depuis la côte d'un cargo au large de Singapour [30]. Les services postaux ont également reconnu une innovation affectant directement leur domaine métier ; en Suisse par exemple, Swiss Post expérimentant des drones américains de Matternet depuis 2015 [8]. Ces moyens de transports sont appréciés pour leur vitesse, leur faible coût, leur faible bilan carbone, et la possibilité qu'ils offrent de rejoindre des zones difficiles d'accès ou recluses du fait d'une catastrophe naturelle. La rapidité de livraison en font également des véhicules de choix pour transporter des ressources ou du matériel médical. Nous avons préalablement évoqué

les entreprises qui livrent des poches de sang ou des médicaments ; parmi les autres usages apparaissent les drones ambulances, permettant de convoquer un défibrillateur [189, 140].

1.2.5 La secteur de la sécurité

Les drones peuvent être rapidement déployés pour surveiller de vastes étendues difficiles d'accès. Dans le domaine de la sécurité, cela laisse entrevoir des baisses de coûts en diminuant les effectifs nécessaires à la surveillance. Pour ces applications, quelques aspects techniques sont à prendre en compte ; les critères d'optimisation comprennent en premier lieu l'autonomie et la robustesse. Pour l'autonomie, cela fait également envisager des solutions à moteurs à combustion et explosion, le poids des batteries électriques restant un facteur limitant (voir [58] pour une revue des méthodes d'alimentation d'un drone). La robustesse est nécessaire pour pouvoir opérer le drone dans différentes conditions météorologiques. A l'instar de la gestion d'infrastructures, deux cas sont à distinguer : la surveillance d'un site de faible surface et la surveillance d'une région étendue, telle qu'une frontière ou une côte. Dans le premier cas, des dispositifs d'alimentation complémentaires sont envisageables, comme une alimentation continue par câble d'un drone captif ou par laser.

Les applications sont nombreuses : surveillance des frontières, lutte contre braconnage, *etc.* Les drones permettent également de surveiller les feux de forêt ; équipés de capteurs infrarouges, ultraviolet, et optiques, ils tracent l'origine et l'évolution des incendies en fournissant des informations sur les foyers des fournaises. L'ensemble de ces utilisations se démocratise : en 2014, la port d'Abu Dhabi s'était déjà équipé en drone pour la surveillance. Pour la coupe du monde de football au Brésil en 2014, le pays hôte avait acheté des drones israëliens pour surveiller les foules [27]. En France, les forces de l'ordre en déploient pour différentes opérations de patrouille, par exemple pour vérifier le respect des zones hors limites telles que les plages lors du confinement lié à l'épidémie de COVID-19 [29]. Cet usage pour la surveillance du respect des règles de sécurité sanitaire a toutefois été proscrit par le Conseil d'État, le jugeant comme une atteinte grave au droit au respect de la vie privée du fait des capacités d'identification des personnes [68].

1.2.6 Le secteur anti-drone

Les drones ont été utilisés pour différents coups d'éclat, à l'instar de vols près de personnalités politiques. Cela a renforcé l'émergence d'un marché anti-drones pour faire face à ce nouveau danger aérien. Différents moyens sont envisagés, des plus technologiques aux plus rudimentaires. L'armée française a ainsi fait l'acquisition de quatre aigles royaux en 2017 qui sont entraînés contre les drones [31]. Ces solutions sont envisagées dans des environnements urbains. La police néerlandaise a fait un choix similaire.

Battelle a produit DroneDefender, un système d'interception à distance de drone [6]. D'autres systèmes permettent de détecter un drone et son télé-pilote, tels que l'AeroScope de DJI. Il repose sur le brouillage des communications entre le télé-pilote et l'engin, ce qui reste pour l'instant interdit par des législations à l'instar de celle américaine. La plupart de ces législations interdisent d'autre part les systèmes destructifs pour l'interception de drone.

1.3 La législation

Jusqu'à présent, la législation du drone dans l'Union européenne était laissée aux états membres. En France, les contraintes étaient données selon la catégorie de l'opération drone. Les principales caractéristiques des différents scénarios sont présentées dans le tableau 1.1. Ces

scénarios s'accompagnent d'obligations, plus lourdes pour les professionnels que pour les pilotes de loisir (en terme d'assurance ou d'immatriculation par exemple). À ces catégories s'ajoutent les vols expérimentaux, pour lesquels des dérogations peuvent être obtenues, mais impliquent une liaison étroite avec la Direction de la Sécurité de l'Aviation Civile.

TABLE 1.1 – Les différents scénarios pour l'opération de drone de la réglementation française qui était jusqu'alors en vigueur. Le scénario dépendait de l'altitude maximale, de la distance maximale par rapport au télé-pilote, du poids du véhicule, de la présence de personnes sur la zone survolée, et du maintien d'une ligne de vue entre le télé-pilote et le véhicule

S1	150m	200m	25kg	Non	Vol à vue
S2	50m	1km	25kg	Non	Vol en immersion
S3	150m	100m	4kg	Oui	Vol à vue
S4	150m	∞	2kg	Non	Vol en immersion

Le 4 juillet 2018, l'Union européenne est parvenue à un nouveau cadre réglementaire s'appliquant aux domaines de l'aviation civile, et incluant les drones. Les États membres sont tenus de faire entrer cette réglementation en vigueur d'ici juillet 2020 [5]. Pour un résumé de cette réglementation du point de vue droniste, nous renvoyons vers [46]. Parmi les contrariétés restantes, la protection de la vie privée demeure un défi principal. Une étude pour le Parlement européen [89] avertie ainsi que les images captées par les caméras dédiées au télé-pilotage peuvent être enregistrées, et potentiellement mises en circulation sur internet. « L'enregistrement par des drones d'images de personnes se trouvant dans leur maison ou dans leur jardin peut constituer une atteinte à l'intimité de la vie privée et de la propriété privée, ainsi qu'une violation des droits des citoyens en la matière ». D'autres applications et capteurs sont également à prendre en compte, à l'instar de la reconnaissance faciale, de la détection de mouvements, des capteurs thermiques, des capteurs Wi-Fi, des microphones. En 2015, l'étude arguait que l'environnement technologique permettant une intégration des drones dans le système de l'aviation civile répondant aux enjeux de sûreté et de sécurité n'était pas encore connu. De plus, les opérateurs ne sont pas toujours sensibilisés aux problématiques d'atteinte à la vie privée et de protection de données que soulèvent leurs usages des drones [98]. Ces considérations amènent toutefois des constructeurs tels que DJI ou Parrot à chiffrer les journaux d'exécution pour empêcher leur dissémination. Des services internes aux entreprises peuvent produire un rapport sur ces données en cas de crash, mais l'analyse des traces d'exécutions n'en est que complexifiée pour l'opérateur.

La législation européenne classifie dorénavant les opérations drones en trois scénarios :

1. *ouvert*, catégorie comprenant de faibles risques
2. *spécifique*, catégorie intermédiaire, nécessitant une analyse au cas par cas
3. *certifié*, catégorie présentant des niveaux de risques similaires à l'aviation traditionnelle

En l'état actuel, les usages commerciaux de drones tombent dans la catégorie spécifique. Pour celle-ci, l'analyse de risques peut [6] être conduite selon SORA, la méthode promulguée par les *Joint*

5. Dans le contexte de la pandémie COVID-19, la date d'entrée en vigueur a été repoussée au 1er janvier 2021 [69]

6. Bien qu'il n'y ait aucune obligation, cette méthode est fortement encouragée. De plus, l'habitude que développeront les autorités régulatrices ne manquera pas de convaincre la majorité des opérateurs d'y recourir. Elle peut toutefois être complétée par d'autre méthodes, telles qu'une Analyse des Modes de Défaillance, de leurs Effets et de leur Criticité

Authorities for Rulemaking of Unmanned Systems [119]. SORA (pour *Specific Operations Risk Assessment*) est une méthode de gestion de risques, simplifiée, dédiée à une évaluation qualitative des risques liées à la sûreté pour les opérations spécifiques. SORA utilise le concept de robustesse associée à la mitigation des risques ou aux objectifs opérationnels de sûreté, qui est déterminée par la conjugaison d'un niveau de protection (gain en sûreté) et d'un niveau d'assurance (preuve apportée). La robustesse est jugée selon trois niveaux d'assurance :

- *bas*, l'opérateur déclarant simplement que le niveau de protection requis est atteint
- *moyen*, l'opérateur présentant des gages qui attestent du niveau de protection déclaré
- *haut*, une tierce partie réputée compétente ayant validé le niveau de protection déclaré

De même, SORA introduit trois niveaux de protection par mitigation ; les annexes du document SORA guident l'évaluation des niveaux de protection et d'assurance pour la mitigation des risques terrestres et aériens. Finalement, la robustesse est définie comme le minimum des niveaux de protection et d'assurance atteints (cf. tableau 1.2)

TABLE 1.2 – Évaluation de la robustesse en fonction des niveaux de protection et d'assurance dans le cadre d'une analyse SORA

	Basse protection	Moyenne protection	Haute protection
Basse assurance	Basse robustesse	Basse robustesse	Basse robustesse
Moyenne assurance	Basse robustesse	Moyenne robustesse	Moyenne robustesse
Haute assurance	Basse robustesse	Moyenne robustesse	Haute robustesse

Cette étude de la robustesse doit être conduite sur les risques terrestres et aériens : le type d'opération – déterminé par la dimension du drone, l'énergie cinétique attendue, la distance séparant le drone du télépilote, et la densité de population au sol – fournit une valeur initiale catégorisant le risque, qui pourra être diminuée (ou augmentée) selon la robustesse des éléments de mitigation prévus. De là, un SAIL final (*Specific Assurance and Integrity Levels*) est calculé, qui affecte les Objectifs de Sûreté Opérationnelle ; nous renvoyons vers le document JARUS [119], et plus particulièrement vers l'annexe E pour le détail.

Bien que moins contraignante que la réglementation de l'aviation traditionnelle du fait de son caractère informel et qualitatif – ce qui est un motif de critique [79]–, la méthode SORA reste un carcan complexe pour l'opération drone : le temps nécessaire à l'obtention de laissez-passer est conséquent par rapport au temps de développement de nouvelles solutions. La communauté droniste suit de près les évolutions de cette législation, elle propose des exemples d'analyse SORA en guise de partage d'expérience [61, 134], et des systèmes pour assister l'opérateur dans ses démarches émergent [187, 188].

1.4 Un cadre pour la définition de mission

Le drone civil partage ainsi des caractéristiques d'une nouvelle technologie, en particulier des applications commerciales variées et qui restent en partie à inventer, une législation émergente, et un avenir annoncé radieux. Pourtant, les aéronefs non habités ont déjà une histoire d'un siècle dans le domaine militaire, dont un demi-siècle d'usage sur théâtres, qui contribue aux investissements et aux développements techniques du secteur. En conséquence, les acteurs principaux de la scène du drone militaire – *viz.* les Etats-Unis, Israël, et la Chine – disposent d'un avantage concurrentiel. Le prix de l'amorce historique par le militaire se trouve sans doute dans la perception qu'en a le grand public. Cette controverse est de plus alimentée par des opérations « coup de poing » dans le civil, des drones survolant des sites sensibles ou menaçant des personnalités po-

litiques. Cela a considérablement ralenti les initiatives de zones pour l'expérimentation drone en même temps qu'une réglementation précise tardait à venir. Ce manque de visibilité législative est d'ailleurs pointé comme le principal frein à l'expansion du marché. L'obtention d'un laissez-passer pour effectuer des opérations complexes, à l'instar d'une navigation autonome, reste souvent une épreuve rédhibitoire. Le drone civil a une large communauté à l'esprit *makers*, impatiente, qui se heurte à des autorités généralement rompues à l'aviation traditionnelle avec ses processus longs et sa gestion des risques vétilleuse.

De cette situation nous en retiendrons quelques exigences pour notre méthode de définition et de vérification de missions. La diversité des applications existantes, et plus encore de celles potentielles et futures, appelle une expressivité forte. L'introduction de nouveaux éléments, à l'instar de capteurs ou de contraintes, serait compromise dans un système ayant des structures élémentaires déjà spécialisées. Au contraire, il nous apparaît pertinent d'inclure des moyens pour définir les systèmes qui nous intéressent et pour programmer des missions conférant une autonomie au drone.

Une deuxième piste pour notre travail concerne la sûreté de fonctionnement. L'environnement physique étant par essence incertain, il conviendra de prévoir l'ajout de mécanismes de mitigation à nos missions ; suivant l'un des besoins identifiés, nous illustrerons régulièrement notre propos par des propriétés de *geofencing*. L'analyse SORA demande de plus des gages relatifs à ces protections, et il nous semble important d'intégrer la production de ces arguments à la définition de mission. De fait, l'automatisation du passage entre les différentes parties d'une chaîne outillée réduit le risque d'introduction d'erreurs d'origine humaine.

Enfin, le premier cadre que nous esquissons laisse présager d'un système s'adressant à des experts. Il convient dans ce cas de travailler à l'accessibilité de l'outil de définition de mission, afin de l'introduire auprès des acteurs d'une opération drone ne disposant pas d'une vaste expérience en programmation. Une mission gagnerait à produire des supports pensés pour la communication entre les différentes parties prenantes.

Notons enfin que ces considérations, bien qu'établies par l'analyse du marché du drone, sont pour l'essentiel partagées par l'ensemble des systèmes cyber-physiques mobiles, si bien que les exigences que nous en avons extraites ne sauraient être jugées réductrices ou spécifiques. Nous verrons même qu'un bref examen épistémologique sur la signification des systèmes cyber-physiques renforcera plusieurs de nos réflexions particulières, au rang desquels la pluralité des acteurs, et la diversité et la criticité des opérations ; ce sera l'objet de notre prochain chapitre.

Chapitre 2

Les systèmes cyber-physiques et leur programmation

Les systèmes cyber-physiques recouvrent de nombreuses disciplines et objets d'étude. Nous en avons vu une instance dans le premier chapitre avec les drones, sur lesquels notre intérêt portera principalement dans la suite de cet ouvrage. On pourra à cet égard considérer les drones dans un sens moins restrictif que celui utilisé jusqu'ici, à savoir comme tout système robotique mobile autonome ou télé-piloté, dont les aéronefs non habités ne sont qu'une catégorie. Les systèmes cyber-physiques sont des entités doubles, comprenant un contrôleur logique et des composants matériels permettant des interactions avec le monde physique. Bien qu'elle suggère nombre des caractéristiques qui nous occuperont le restant de cet ouvrage, cette première définition demeure toutefois assez approximative. En effet, contrairement aux drones les systèmes cyber-physiques désignent moins des objets concrets qu'une manière de les étudier. C'est tout du moins la thèse qui apparaîtra de l'analyse épistémologique que nous mènerons dans la première partie de ce chapitre. Nous verrons que, sans les reprendre point par point, les exigences qui ont émergé dans le premier chapitre et qui portaient sur les drones (fiabilité, analyse multi-domaine, diversité des applications et des acteurs) transparaissent plus largement dans l'étude des systèmes cyber-physiques. Nous nous intéresserons alors à la programmation des systèmes cyber-physiques mobiles, avec une attention particulière portée sur le secteur du drone.

2.1 Que sont les systèmes cyber-physiques ?

Les système cyber-physique constituent aujourd'hui un vaste ensemble d'outils qui traversent tous les domaines de la société et de l'industrie. Il apparaît difficile de définir précisément ce que sont les systèmes cyber-physiques. Selon les auteurs, ils recouvrent, au moins partiellement, un grand champs d'objets tels que ceux traités par la robotique, la domotique, les systèmes embarqués. Cependant, l'objet statique ne suffit pas à définir la notion de système cyber-physique. D'autres considérations rentrent en jeu, en particulier des analyses non-fonctionnelles tels que l'usage, la sûreté de fonctionnement ou l'inclusion dans un réseau et les problématiques de sécurité de l'information, bien qu'aucun de ces thèmes ne soit indispensable.

Les ouvrages traitant des systèmes cyber-physiques choisissent généralement de donner une intuition de ces objets sans les définir. Dans [166] par exemple, l'angle d'approche dans la préface consiste à les opposer aux systèmes logiciels traditionnels. La caractérisation donnée des systèmes cyber-physiques est la collaboration d'éléments informatiques pour contrôler des entités physiques qui interagissent avec des humains et leur environnement. Pour les différencier

des systèmes étudiés par la théorie du contrôle traditionnelle, les auteurs notent les problématiques de sensibilité au contexte, d'informatique cognitive, et d'autonomie. Dans l'avant-propos du même ouvrage, Koopman, parlant des systèmes embarqués, écrit : « At some point these complex systems become Cyber-Physical Systems (CPS), which in general involve multiple computer systems interacting to sense and control things that happen in the real world ». Là encore, aucune caractérisation ontologique ne distingue les systèmes cyber-physiques d'autres systèmes ; mais la différence d'échelle (sur un axe qu'il reste à définir) laisse entrevoir un outillage distinct pour leur conception et leur analyse, ce qui n'est pas sans rappeler la distinction faite entre le *big data* et le traitement de données traditionnel. Il nous apparaît opportun de formuler cette distinction suivant les dimensions dénotatives et connotatives. En effet, rappelons que la signification est une mise en relation d'un plan d'expression et d'un plan de contenu. Barthes, dans le lignée de Hjelmslev, considère alors un système de signification dont le plan d'expression est lui-même un signe et l'oppose au système primaire qu'est ce signe. Il le nomme *mythe* par opposition à *langue* [44], ou *connotation* par opposition à *dénotation* [45] (cf figure 2.1). La dénotation est ce que l'on considère généralement comme le sens premier, ou littéral, d'un terme, tandis que la connotation correspond à « [t]out ce que ce terme peut évoquer, exciter, impliquer de façon nette ou vague chez chacun de ses usagers » pour reprendre une définition de Martinet [132]. Le terme système cyber-physique nous semble dépourvu d'une sémantique dénotationnelle précise, mais présente au contraire une sémantique connotationnelle prépondérante.

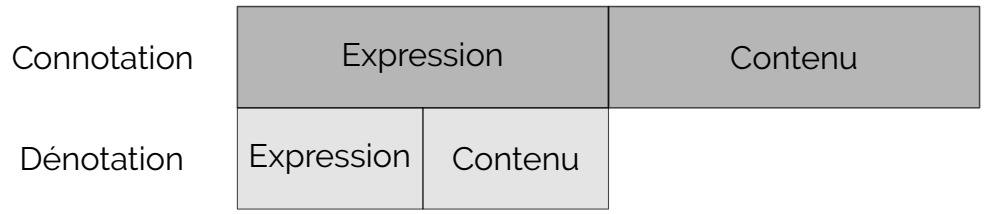


Figure 2.1 – Les plans de dénotation et de connotation

Une même réalité se cache souvent derrière des termes aussi variés que l'internet des objets, les communications inter-machines ou les systèmes cyber-physiques [118, 105]. Bien que les définitions semblent converger, ou, du moins, que des auteurs et institutions appellent à cette convergence, des nuances persistent encore dans les esprits. En substance, nous affirmons ici avec Eco « qu'il n'existe pas de véritables synonymes : lorsque le même signifié est apparemment exprimé par deux signifiants différents, nous sommes en réalité toujours en présence de nuances différentes [...] » [96]. Le choix du terme dépend alors d'une attitude, d'une approche. Ainsi, devant une chaudière contrôlée depuis un smartphone pour réguler la température d'une pièce dotée de thermomètres, l'expert réseaux parle d'objets connectés quand il s'intéresse aux protocoles de communication tandis que l'automaticien pense système cyber-physique en implantant sa loi de contrôle. La préséance donnée à la dénomination système cyber-physique dans cet ouvrage doit donc servir d'avertissement quant au traitement de ces objets : l'étude porte ici sur le contrôle d'une entité considérée atomique plutôt que sur l'aggrégation des composants qui la constitue.

Cette dernière présentation de ce qu'est un système cyber-physique nous inscrit dans une longue tradition de définition pragmatique, autrement dit de définition par l'usage. Wittgenstein y recourt dans ses *Investigations philosophiques* (remarque 43) pour définir le « sens » d'un mot : « Dans une catégorie considérable de cas – mais non pour tous – où nous employons le terme « sens », on peut le définir ainsi : le « sens » d'un mot est son emploi dans la langue » [193].

Heidegger a d'ailleurs eu préalablement une vision plus universelle de cette pragmatique, rappelant par exemple que c'est l'usage de l'outil qui en fait un outil (un outil *s'utilise*) [112]. Dès lors, dans cette acception, définir un système cyber-physique comme fonction de son environnement (dont l'observateur n'en est qu'une partie), en somme comme d'une fonction de son usage, n'est qu'affirmer une lapalissade. La question devient de savoir quel(s) usage(s) font le système cyber-physique.

La relation étroite entre un système cyber-physique et le contrôle que nous avons esquissée est une caractérisation classiquement retenue (voir par exemple [62, 131, 182]). Du reste, les thermomètres ou la chaudière du système thermostatique évoqué *supra* pris séparément ne seront pas considérés comme des systèmes cyber-physiques. Dans notre acception du terme, deux transducteurs (capteurs et actuateurs) et une fonction logique reliant la commande à la mesure sont requis. L'asservissement d'un processus physique, en ce qu'il correspond à un système commandé par rétroaction, constitue donc une première théorie des systèmes cyber-physiques. Formellement, Knight *et al.* [123] définissent un système cyber-physique comme un tuple de trois éléments :

- un ensemble d'« entités du monde réel » avec lesquels le système interagit
- une « plateforme matériel », qui exécute le logiciel du système cyber-physique et fournit les connexions physiques entre les entités réels et le logiciel
- un « formalisme interprété », mêlant définition du logiciel et son interprétation

Cette définition, dans son choix des termes, nous semble toutefois introduire une certaine simplification, en faisant fi du caractère inexorablemment empirique de ces objets d'études. De même que Paty avertissait du hiatus entre la théorie d'une science naturelle et un formalisme interprété [147], les injonctions physiques qui sous-tendent cette discipline en en précédant la théorie en font une composition de concepts formalisés plutôt qu'un formalisme interprété [7]. En conséquence, une comparaison de théories de système cyber-physique est alors possible, par un principe de correspondance, en les confrontant aux contenus sous-jacents. Ces contenus sont de plus extrêmement variés. La conception, la modélisation, et l'analyse de système cyber-physique est une discipline d'une grande richesse par la diversité des domaines qu'elle traverse. Pour nous en convaincre, observons une liste (non exhaustive) des champs de la connaissance qui sont impliqués dans notre exemple de système thermostatique : traitement du signal, électronique, thermodynamique, combustion, théorie du contrôle, réseaux. L'élaboration d'un tel système est donc un problème multi-métiers, et nécessite une pensée holistique. Cela a des conséquences à tous les niveaux. La simulation, par exemple, ne peut plus être abordée comme un problème mono-discipline, mais requiert au contraire de mettre en relation les différents outils dans une vaste co-simulation [145]. Par ailleurs, les systèmes cyber-physiques affectent le monde réel et constituent en cela des systèmes critiques (naturellement, le niveau de criticité d'une pompe à insuline sera bien plus grand que celui de notre thermostat). Dans leur construction et dans leur usage, il convient donc de prendre en compte d'autres acteurs, tels que les autorités régulatrices.

La modélisation des systèmes cyber-physiques insiste généralement sur les interactions bi-directionnelles entre processus physiques et computations, dont l'étude requiert l'utilisation conjointe des dynamiques discrète et continue. Les auteurs s'intéressant à la dynamique de ces systèmes (par opposition à une analyse statique, *viz.* structurelle) en font d'ailleurs le marqueur de la discipline [80]. Platzer [153] les définit tant par le couplage que par les outils pour leur analyse : « Cyber-Physical Systems combine cyber capabilities such as communication, computation and control with physical capabilities such as the motion of robots, cars, or aircraft. Mathematical models for such CPS are based on hybrid systems, which combine discrete dynamical

7. Nous reviendrons sur ce point et ses conséquences pour nos travaux au chapitre 3.

systems with continuous dynamical systems, e.g., because discrete change one step at a time fits well to computation, while continuous dynamics along differential equations fits well to their motion ». C'est là aussi ce que retient [166] : « The main characteristics of physical systems are their continuous behavior and stochastic nature. This is why the development of CPS requires the involvement of specialists in multiple engineering disciplines. These systems often use close interaction between the plant and the discrete event system, and their complexity can grow due to interaction among multiple CPS and the need to deal with time. ».

De la boucle capteurs - logique - actuateurs, le reste de l'ouvrage traitera principalement du composant central. Nous écrirons sans distinction fonction logique, contrôleur, processus décisionnel, programme. Évidemment, ces derniers termes semblent emphatiques pour un régulateur PID, mais ils prennent tout leur sens à mesure que la fonction logique se complexifie jusqu'à fournir une autonomie complète pour un système aussi complexe qu'un drone, ou qu'une « ville intelligente ». Les travaux présentés dans la deuxième partie de cet ouvrage ne prétendent pas insuffler une autonomie dans son sens le plus noble de principe téléologique. Ils ambitionnent toutefois d'automatiser l'usage des systèmes cyber-physiques par la programmation.

2.2 La programmation de systèmes cyber-physiques

2.2.1 En robotique générale

La programmation téléo-réactive [144] conditionne le comportement d'un agent à sa perception de l'environnement. Pour cela, une séquence téléo-réactive est une séquence ordonnée de règles de production, associant à une condition liée aux capacités sensorielles de l'agent une action affectant le monde. L'ordre de ces règles entraine une évaluation en cascade ; la disjonction de l'ensemble des conditions d'une séquence téléo-réactive complète étant une tautologie. Ce formalisme est similaire à bien des égards à notre langage : les règles téléo-réactives pourront ainsi être comparées à nos contextes. La différence principale est la vision automate (ou circuit, du fait des règles de production) d'une séquence téléo-réactive tandis que `Sophrosyne` adopte un point de vue plus impératif en introduisant également des outils communs des langages de programmation tels que des blocs d'instructions, et des mécanismes pour contrôler explicitement le flux d'exécution.

Le langage `Maestro` [72] était développé pour la définition de mission robotique pour le système `Orccad` [57, 175]. Ce dernier présentait deux structures clés, les *Robot-Tasks* et les *Robot-Procedures*. Les *Robot-Tasks* sont des lois de contrôle atomiques, exprimées dans le domaine continu et contenant des événements discrets qui surviennent lors de l'exécution. Les *Robot-Procedures* sont des compositions logiques de *Robot-Tasks*, de manière séquentielle, avec des boucles, de l'exécution parallèle, ou de la préemption. Les contrôleurs robotiques étaient alors compilés en `Esterel` afin de vérifier formellement les automates qui en résultaient. Le logiciel permettait de plus d'effectuer des simulations logiques du contrôleur afin de pallier les limites de la preuve formelle, notamment sur les systèmes hybrides. Nous retrouvons ici la séparation entre des actions primitives et leur utilisation dans des structures à complexité croissante que nous qualifions de missions. Notre approche diffère toutefois en ce que nous distinguons l'exécution réelle de l'exécution idéale, ce que nous développerons au chapitre suivant.

Le langage dédié `P` [82] permet d'écrire des programmes événementiels asynchrones. Pour cela, un ensemble de machines à état interragissent en s'envoyant des événements. Le système comprend également un *model-checker*, `Zing`, pour vérifier les programmes. `Drona` [84, 83] étend `P` avec des capacités de vérification de l'exécution, pour la programmation de systèmes robotiques mobiles distribués. Les programmes sont alors compilés en `C` et exécutés dans `ROS`.

`ROS` (*Robot Operating System*) [20] est un ensemble d'outils *open source* permettant de développer des logiciels pour la robotique. Il s'agit d'un intergiciel qui peut être vu comme un méta-système d'exploitation qui peut être distribué sur plusieurs machines et qui offre des fonctionnalités d'abstraction du matériel spécifique à la robotique. En premier lieu, il est en son coeur un service de transmission des messages « publish-subscribe » pour les applications qui sont développées au-dessus. Ces applications peuvent être développées dans divers langages de programmation avancés comme `C/C++` ou `Python`. Une de ses forces est sa grande communauté d'utilisateurs, qui se traduit entre autres par la présence de très nombreux pilotes de matériels. Autre fonction d'intérêt pour le cycle de vie des applications développées est l'interfaçage facilité avec des moteurs de simulation (monde physique et visualisation), interne avec `RViz` ou externes avec *e.g.* `Gazebo` [9] ou `CoppeliaSim` [5]. Le passage à `ROS2` [21] complète cet écosystème notamment avec des aspects de qualité de services (garantie temps-réel si les systèmes sous-jacents le permettent).

2.2.2 Pour les drones

L'essor du drone que nous avons vu au chapitre 1 s'est naturellement accompagné d'un intérêt croissant pour leur programmation.

Les constructeurs de drones (plus particulièrement d'autopilotes) grand publics fournissent généralement un logiciel pour leur utilisation. Celui-ci, avant de permettre la définition de mission, reste nécessaire pour paramétrer et calibrer l'autopilote, ainsi que pour disposer d'un retour sur l'état du drone pendant le vol. Ces fonctionalités en font des logiciels de station de contrôle au sol dont les principaux représentants du marchés sont QGroundControl [158], MissionPlanner [39], DJI GS Pro [90], et les applications Parrot FreeFlight [146] pour un usage sommaire, et Pix4D [149] pour les utilisations de photogrammétrie. Concernant la composante mission, leur expressivité, bien que croissante, est encore limitée. Une mission est considérée comme une liste de *waypoints* (points de passage, auxquels sont parfois associées des orientations et des vitesses). Quelques fonctionnalités de haut niveau permettent de générer ces *waypoints* selon des patterns, notamment pour l'inspection d'une surface (un exemple avec QGroundControl est donné par la figure 2.2). L'utilisateur peut de plus modifier l'attitude de la nacelle, ajouter des *geofences* pour restreindre la zone de vol, définir des points de ralliement pour des atterrissages automatiques d'urgence, et configurer les changements de mode (par exemple en cas de perte de signal radio, de franchissement de *geofence*, ou de charge insuffisante de la batterie).

Les logiciels de station de contrôle au sol supportent généralement deux catégories de *geofences* : celles circulaires, et celles polygonales. Dans les deux cas, la troisième dimension est contrainte séparément par une limite sur l'altitude maximale du drone. Les *geofences* ainsi définies sont statiques : elles sont actives tout au long du vol. Le paramétrage de l'autopilote permet de spécifier le seuil de tolérance (durée que peut passer le drone hors de son volume de vol avant que la sécurité ne soit déclenchée) et le comportement à adopter en cas de franchissement de la *geofence*. Ce dernier paramètre se résume à un changement de mode de l'autopilote, tel qu'un maintien de la position ou un retour à la base. Les solutions les plus avancées offrent la possibilité d'associer un point de ralliement à chaque *geofence*, s'apparentant à des zones d'atterrissage d'urgence.

Plusieurs de ces logiciels reposent en définitive sur le protocole de référence `MAVLink` (*Micro Air Vehicule Link*) [12] pour communiquer avec l'autopilote. Il s'agit d'un protocole simple et léger conçu pour communiquer avec un drone, voire entre drones. Principalement destiné à la communication point-à-point entre la station de base et les drones, il peut toutefois être utilisé pour la communication entre les différents sous-systèmes qui composent le drone. Il dispose

FIGURE 2.2 – Pattern *survey* pour la définition de mission avec QGroundControl

par exemple de messages décrivant l'état du drone (*e.g.* position, vitesse, orientation) ou de commandes (*e.g.* atterrir, décoller, se rendre à un point géographique). Ces commandes sont toutefois assez limitées : de cela procède la faible expressivité des stations de contrôle au sol décrites précédemment. Pour accroître la complexité des missions, le programmeur tire profit de *Software Development Kit* écrit au-dessus de `MAVLink` tels que `MAVSDK` [13] ou `DroneKit` [7]. Il écrit alors dans des langages de programmation avancés (`Python`, `C/C++`, *etc.*) des applications s'exécutant sur un ordinateur de bord ou sur une station au sol. Cette méthode de développement nécessite de fait d'importantes compétences en programmation.

Du côté académique, [171] identifie des limitations de ces logiciels de station de contrôle au sol. L'absence de *waypoints* conditionnels (les *geofences* mises à part) restreint à des missions linéaires. Les auteurs déplorent également l'impossibilité de définir des tâches concurrentes, telles que l'évitement d'obstacles. De plus, certains de ces logiciels sont propriétaires, à l'instar de DJI GS Pro. L'article précise alors les contours d'un langage dédié graphique pour la description haut-niveau de missions de drone pour les utilisateurs non scientifiques, incluant les fonctionnalités précédemment évoquées. FLYAQ [85, 86, 59] est un logiciel *cross-plateform* pour la spécification de missions pour des flottes de drones. Il fournit une famille de langages dédiés pour définir les missions, les comportements des robots, ainsi que les robots eux-mêmes. Les auteurs s'étaient également fixé pour objectif de permettre à des utilisateurs non-experts de décrire des missions complètes. Pour cela, le logiciel intègre par exemple un module de génération partiellement automatique des trajectoires et *waypoints* de mission.

[139] dénonce une autre limitation de ces approches : l'utilisation de *waypoints*. La simplicité de ces éléments, bien qu'appréciable pour leur rapide prise en main, les disqualifie pour la définition de missions complexes. La solution proposée et implémentée dans la suite logicielle Aerostack [167] est un langage de spécification de tâches utilisant une syntaxe XML. L'utilisateur écrit deux fichiers : (i) un arbre de tâches, qui contient les actions à effectuer et les aptitudes que doit présenter le robot pour exécuter une tâche, et (ii) une gestion d'événements, qui spécifie le comportement à adopter dans des situations particulières telles qu'un niveau critique de charge

de la batterie. La faisabilité de la mission est alors vérifiée, prenant en compte les conditions d'exécution des tâches avant et pendant l'exécution de la mission. De même, [168] propose d'introduire des étapes (*legs*), des structures plus expressives que les *waypoints* déjà utilisées dans l'aviation commerciale pour la spécification de navigation de surface. Les étapes étendent la notion de *waypoint* en indiquant la trajectoire à suivre pour rejoindre le *waypoint*. Le planificateur de mission dispose d'une bibliothèque de comportements, comprenant notamment des boucles et des exécutions conditionnelles.

Une hiérarchie de langages formels de description d'intention de vols est présentée dans [129, 53, 100]. Le *(QuadRotor-)Aircraft Intent Description Language* est construit sur la dynamique de l'aéronef. Pour qu'elle soit valide, une intention en AIDL requiert de contraindre tous les degrés de liberté de l'aéronef. L'utilisateur spécifie pour cela des instructions telles que maintenir la vitesse ascensionnelle. La transition d'une instruction à une autre est déclenchée par des événements. La condition peut alors être un marqueur temporel spécifique, une fonction de l'état du véhicule, ou une dépendance envers la transition d'une autre instruction. Des extensions ont été développées autour d'AIDL constituant une hiérarchie de langages. ICDL ajoute à AIDL des mécanismes de composition permettant de paramétriser les intentions et de laisser des degrés de liberté non contraints ; cela permet en particulier de *raffiner* des intentions de vols. FIDL complète cette hiérarchie en incorporant également la définition de contraintes sur la trajectoire et d'objectifs, c'est-à-dire de fonctions à optimiser telles que la durée du vol. Ces travaux ont été appliqués dans le cadre d'un système de définition de missions [52]. L'approche décrite est de rassembler différentes parties prenantes d'une opération drone (client, opérateur drone) autour d'une application présentant diverses fonctionnalités telles que la génération d'une mission drone à partir d'une description haut-niveau de l'opération, une évaluation de la faisabilité de la mission, ou la création de vidéos de simulation de la mission.

2.2.3 Un exemple pour les robots aspirateurs

Nous avons vu précédemment que, dans le cadre de la définition de mission, les principaux logiciels de station de contrôle au sol sont critiqués pour leur faible expressivité. Il nous semble utile de souligner ce point par une comparaison avec une autre catégorie de logiciel de contrôle : la gestion de robot aspirateur. L'application que nous considérons en guise d'illustration est la Mi Home [117] de Xiaomi pour un roborock S5 max (cf figure 2.3).

Le roborock S5 max est un robot aspirateur incorporant également une serpillère. Les fonctionnalités de l'application qui nous intéressent comme points de comparaison sont :

- l'envoi du robot à une coordonnée pour un nettoyage local, par rapport à l'envoi d'un *waypoint*
- la détection de pièces et le nettoyage par pièce, par rapport à la génération de *waypoints* selon des patterns
- la définition de zones interdites et de murs virtuels, par rapport aux *geofences*
- la définition de zones à mode restreint (ne pas passer la serpillère), par rapport aux consignes pour la nacelle

De fait, sur le plan de la définition de mission, les deux applications nous apparaissent essentiellement équivalentes. Pourtant, si l'usage d'un robot aspirateur se limite à nettoyer, les drones sont susceptibles d'effectuer des missions variées (cf chapitre 1). Quant à la sûreté de fonctionnement, les deux système cyber-physique n'ont pas non plus des niveaux de criticité similaires. Le robot aspirateur se meut au sol, dans un espace délimité, privé, suffisamment lentement pour ne pas faire craindre des chocs. Les principaux éléments d'attention sont la sécurité des communications et la détection du vide pour ne pas dévaler du haut d'un escalier. Le drone évolue dans un

environnement 3D, souvent non maîtrisé et public. La chute libre suffit à mettre en danger la vie d'autrui, même si des dispositions sont prises pour réduire l'énergie de l'impact au sol (pensons par exemple à un drone se posant sur une route).

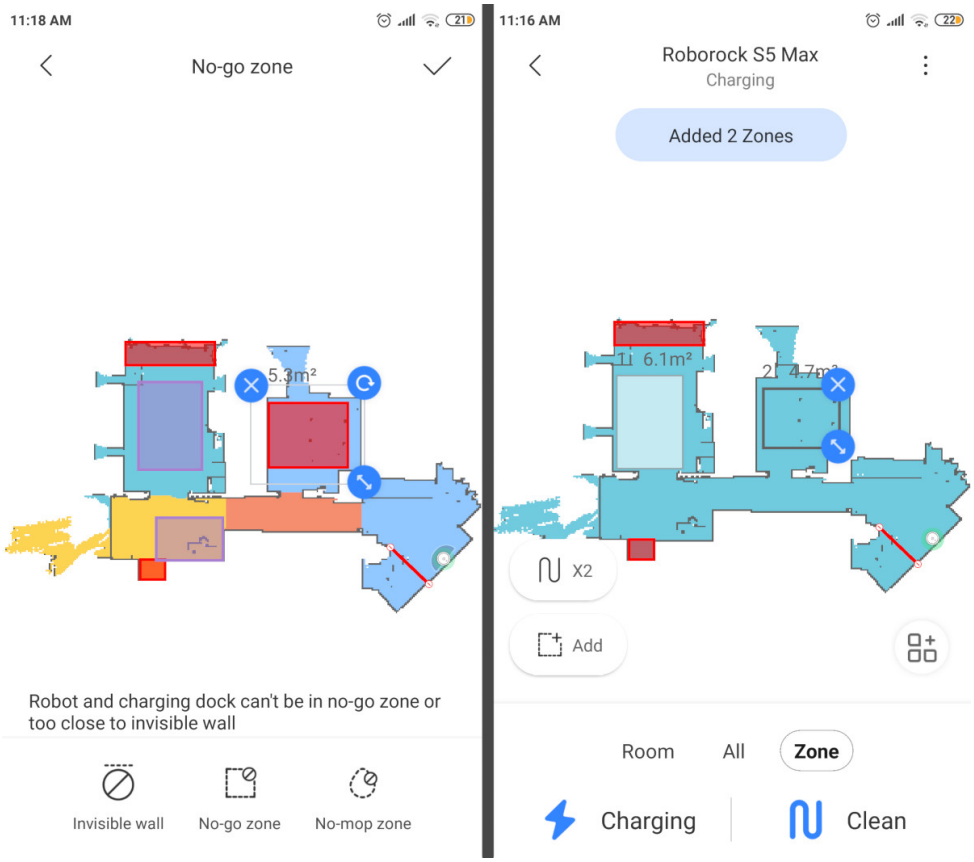


FIGURE 2.3 – L'application Mi Home pour le roborock S5 max permet de définir des pièces à nettoyer, des zones interdites, et des murs virtuels

2.3 Conclusion

Nous avons vu que l'étude des systèmes cyber-physiques impliquait par essence une approche multi-domaine, réunissant divers experts. Au centre de cette approche siège le contrôleur, calculant les commandes transmises aux actuateurs selon les valeurs des différents capteurs et l'avancement dans la mission. Le contrôle se subdivise selon son niveau d'abstraction, de la planification de mission en langage naturel jusqu'à l'asservissement des actionneurs. Dans cet ouvrage, nous nous intéressons principalement à des strates assez hautes dans cette hiérarchie d'abstractions, qui sont à cheval entre la planification de mission et la gestion de la mission à l'exécution.

Les principaux outils commerciaux de définition de mission pour les drones sont peu expressifs : ils sont inadaptés pour les missions complexes que laissent entrevoir les applications vues au chapitre 1, qui comprenent par exemple des exécutions conditionnelles fines. Ce constat a motivé des travaux académiques qui proposent de nouvelles interfaces pour la panification de missions, dont nos contributions que nous présenterons dans la seconde partie sont un exemple. Ils partagerons de fait des traits des différents outils que nous avons exposés : granularité de

la mission plus fine que le *waypoint*, exécution conditionnelle et boucles, analyse théorique des missions, diversité des utilisateurs et parties prenantes. En cela nous retrouvons des conclusions que nous avions tirées de l'analyse du secteur du drone (voir section 1.4). Notre solution, loin d'écarter les logiciels bas niveau dont nous avons présenté quelques parangons pour la programmation robotique à l'instar de `ROS` ou `DroneKit`, s'appuie sur eux ou sur les *API* fournies avec les systèmes considérés pour faciliter et accélérer le développement. L'originalité de notre démarche est la fusion que nous opérons entre la mission et la supervision, qui est usuellement traitée comme un couplage (*e.g.* la spécification de *geofences* est faite indépendemment de la mission). Nous aurons l'occasion de développer cela lorsque nous présenterons le langage `Sophrosyne` (cf. chapitre 4). La deuxième caractéristique de nos travaux, qui n'est certes pas leur exclusivité, est l'approche formelle des missions ; nous allons contextualiser cet aspect dans le chapitre 3.

Chapitre 3

La vérification de systèmes cyber-physiques

In theory, there is no difference between theory and practice. But in practice, there is.

Yogi Berra

L'opération des systèmes cyber-physiques pose des questions quant à la correction des programmes exécutés : le comportement constaté correspondra-t-il aux spécifications ? Il n'est en effet aucunement possible de faire de l'essai-erreur sur le drone réel devant survoler des lignes électriques, tant les conséquences d'échecs pourraient être désastreuses. Ces interrogations sont saillantes dans l'étude de tels systèmes critiques sans en être l'apanage ; diverses méthodes existent pour vérifier un système informatique ou non et accroître la plausibilité de son succès. Les outils de programmation de systèmes cyber-physiques que nous avons présentés dans le chapitre précédant possèdent eux-mêmes occasionnellement des facilités pour la vérification de mission à des niveaux variés, sans que cela ne soit leur première intention. L'objet de ce chapitre est d'esquisser un panorama des méthodes dédiées à la vérification des systèmes robotiques mobiles. Ce domaine est naturellement extrêmement vaste, et dans la pléthore de solutions existantes nous avons choisi d'en présenter une sélection traitant spécifiquement de la vérification théorique. Cette présentation sera précédée par une réflexion sur la notion de modèle : les méthodes que nous présenterons reposent intrinsèquement sur des modèles du système qui en limitent la portée des conclusions. Ce sera aussi l'occasion d'évoquer l'ingénierie dirigée par les modèles dans laquelle s'inscrivent nos contributions. L'exposé des techniques de vérification qui suivra se divisera en deux sections : les méthodes de vérification hors ligne, intervenant à la planification de la mission, puis en ligne, s'intéressant à l'exécution. Nous évoquerons enfin des travaux appliquant des techniques de vérification formelle aux drones.

3.1 La notion de modèle

3.1.1 Qu'est-ce qu'un modèle ?

Il nous faut à présent ouvrir une parenthèse sur une notion dont l'étude mériterait un ouvrage dédié. Celle-ci dépasse néanmoins le cadre de ce travail. Le développement qui suit devrait toutefois éclaircir certains choix de conception et d'usages que nous présenterons plus loin. Il

n'y a certes rien de singulier à introduire un *modèle* afin de raisonner sur les systèmes cyber-physiques. La modélisation est ordinairement considérée comme une activité préalable à l'étude théorique de l'objet.

Minsky définit un modèle d'un objet A comme un objet A' qui permet à un observateur B de répondre en utilisant A' à des questions qu'il se pose sur A [136]. Une spécificité de cette définition est le rôle prépondérant donné à B, l'observateur, et aux questions qu'il se pose. Ainsi, la relation faisant de A' un modèle de A ne peut être évaluée sans B et ses questions. De fait, si une balle rebondissante serait un modèle acceptable pour étudier la durée de la chute libre d'un aéronef, elle n'aurait que peu d'intérêt pour construire un contrôleur en attitude.

Kühne cherche pour sa part à définir le modèle dans le cadre de l'ingénierie dirigée par les modèles [125]. Il parvient alors à la définition suivante : un modèle est une abstraction d'un système (réel ou dans un langage) permettant de faire des prédictions et des inférences. Carreira reprend d'ailleurs cette définition pour parler de multi-modélisation de systèmes cyber-physiques, en précisant que ces prédictions et inférences se font dans un cadre expérimental donné [62]. Ce détail n'est pas sans rappeler l'observateur que nous avons vu dans la définition de Minsky. [179] considère quatre caractéristiques du modèle :

- *l'infidélité* revendiquée à la réalité modélisée, dont la distance est spécifiable (en cela le modèle s'oppose à la théorie qui aspire à être fidèle à la réalité)
- *l'efficacité et l'opérativité* par rapport à des buts déterminés, soulignant la maniabilité du modèle
- *la localité*, le modèle ne s'appliquant efficacement qu'à un ensemble restreint de questions
- *la pluralité des modèles et la cohabitation pacifique*, permettant l'usage de plusieurs modèles d'un même objet ou phénomène

Par rapport aux autres définitions que nous invoquons ici, nous marquons la caractéristique d'infidélité qui présentera des conséquences concrètes dans notre analyse de la sûreté d'une mission `Sophrosyne`.

Dans sa théorie générale des modèles [180], Stachowiak identifie trois attributs d'un modèle :

- la *correspondance*, en ce qu'un modèle représente un original (ou prototype). Celui-ci peut d'ailleurs devenir lui-même un modèle.
- la *réduction*, un modèle ne reprenant pas intégralement les attributs de l'original qu'il représente
- la *pragmatique*, un modèle se substituant à son original dans un cadre précis, à savoir « pour des sujets qui effectuent en des laps de temps donnés des opérations matérielles ou mentales, déterminées par les buts qu'ils poursuivent » [181].

La correspondance doit être vue comme une fonction, une carte projective icostructurale dans les termes de Stachowiak, c'est-à-dire une projection d'un sous-ensemble de prédicats de l'original sur un sous-ensemble de prédicats du modèle. De là, des mesures de la modélisation peuvent être définies comme des différences entre les ensembles d'attributs et ceux concernés par la correspondance, à l'instar de la mesure prétéritive sur l'original ou de celle d'abondance sur le modèle. Stachowiak note qu'inévitablement, les théories scientifiques des objets physiques, psychiques, sociaux, historiques, etc., ont une classe prétéritive et d'abondance non nulle.

Il nous appraît utile de nous positionner par rapport à la Théorie du Système Général de Le Moigne [127], tant les motivations qui président à notre travail peuvent faire écho à ses principes fondamentaux. En effet, nous verrons que `Sophrosyne` partage avec l'activité de modélisation préconisée par Le Moigne l'absence de caractérisation ontologique. Précisons toutefois que la comparaison s'arrête à ces motivations, les ambitions de la Théorie du Système Général dépassant les considérations développées dans cet ouvrage. Le Moigne définit le système comme « un objet qui, dans un environnement, doté de finalités, exerce une activité et voit sa structure in-

terne évoluer au fil du temps, sans qu'il perde pourtant son identité unique ». Cette définition est à lire à la lumière d'un des quatre préceptes du nouveau discours de la méthode qu'introduit l'auteur, le *précepte téléologique* : « interpréter l'objet non pas en lui-même, mais par son comportement, sans chercher à expliquer a priori ce comportement par quelque loi impliquée dans une éventuelle structure. Comprendre en revanche ce comportement et les ressources qu'il mobilise par rapport aux projets que, librement, le modélisateur attribue à l'objet. Tenir l'identification de ces hypothétiques projets pour un acte rationnel de l'intelligence et convenir que leur démonstration sera bien rarement possible. ». Si, de même, nous ne chercherons pas à expliquer le comportement du système cyber-physique lorsqu'il effectue des actions primitives, nous nous refusons à l'utilisation du terme *téléologique*. Le Moigne nomme ainsi son précepte par opposition au principe de causalité cartésien. Dans notre travail, nous préférons présenter cette démarche comme la distinction entre la cinématique qui étudie le mouvement et la dynamique qui en étudie les causes. Il serait inexact d'y voir une réfutation de l'existence d'une causalité, quand nous renonçons simplement à son énonciation que cela soit par méconnaissance ou par simplification. Il conviendra au contraire de penser cette attitude comme une mise en oeuvre de la réduction au sens de Stachowiak.

3.1.2 L'Ingénierie Dirigée par les Modèles

La complexification des logiciels a amené à considérer des méthodes d'abstraction pour leur conception. Au début des années 2000, le *Object Management Group* a ainsi prôné l'architecture dirigée par les modèles (MDA), dont l'idée directrice est de séparer la spécification fonctionnelle du système de la spécification de l'implémentation de ces fonctions sur une plateforme spécifique. Pour ce domaine le terme modèle revêt une dimension singulière, bien qu'il partage de nombreux traits de la notion plus générale que nous avons considérée précédemment. Les notions de base qui encadrent cette acception de « modèle » sont reproduites dans la figure 3.1. L'exemple canonique associé est celui de la syntaxe des langages de programmation. Un programme (le modèle) est conforme à la syntaxe du langage, présentée par exemple sous sa forme de Backus-Naur (méta-modèle). Ce programme permet de représenter chaque exécution (les systèmes), qui siègent elles dans le concret. La grammaire sous forme de Backus-Naur possède elle-même une syntaxe, et est conforme à la grammaire de Backus-Naur sous forme de Backus-Naur qui est conforme à elle-même : le méta-méta-modèle est conforme à lui-même.

Le passage de l'ingénierie orientée objets à l'ingénierie dirigée par les modèles se caractérise par un changement de paradigme : le logiciel passe de « tout est objet » à « tout est modèle ». [54]. La finalité des modèles était jusqu'alors explicative : ils servaient à documenter le code ou d'explication pour son processus de production. La « révolution » de l'ingénierie dirigée par les modèles est de les considérer comme un maillon de la chaîne de production. Cette intégration devient naturelle et explicite lorsque l'on ajoute la génération automatique de code à partir des modèles.

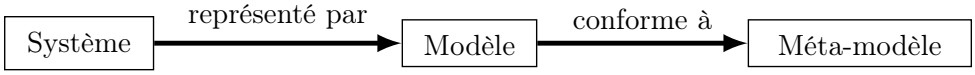


FIGURE 3.1 – Notions de l'ingénierie dirigée par les modèles (adaptée de [54])

L'architecture dirigée par les modèles telle que présentée initialement concentre son attention sur l'axe d'abstraction détachant le modèle de la plateforme d'exécution. Kent [121] note que cette approche tend à considérer chaque modèle dans un formalisme ou langage différent, or différents niveaux d'abstractions peuvent être exprimés au sein d'un même langage. Il s'agit par exemple

du concept de raffinement qu'emploient les méthodes formelles. De plus, il souligne que d'autres axes d'opposition sont possibles, à l'instar d'une perspective par sujet dans la programmation orientée sujet, ou par *aspect* dans la programmation orientée aspect. Ces considérations nous semblent d'autant plus pertinentes pour nos travaux que nous avons montré que l'étude des systèmes cyber-physiques impliquait divers acteurs et expertises.

Les langages dédiés sont des constructions naturellement associées à l'ingénierie dirigée par les modèles. En donner une définition précise est une tâche impossible tant leur contour est flou. En effet, l'opposition faite entre langages dédiés et langages traditionnels [8] semble assez artificielle par rapport à d'autres caractérisations des langages de programmation telles que la complétude au sens de Turing. Notons tout de même que les langages dédiés sont pensés pour répondre aux problèmes d'un domaine d'application. Du point de vue de leur implémentation, ils se divisent en deux grandes catégories, selon qu'ils s'inscrivent dans un langage hôte ou qu'ils soient indépendants [38]. L'inscription dans un langage hôte peut revêtir différentes formes. La plus simple considère une bibliothèque comme un langage dédié : les objets ou structures fournies par cette bibliothèque constituent un vocabulaire adapté à son domaine d'application. Autrement, un langage dédié embarqué peut être défini en réutilisant les outils du langage hôte (*e.g.* les analyseurs lexicaux et syntaxiques). Une dernière catégorie de langage dédié développé sur un langage hôte se fait par l'extension des capacités du langage. `TOM` par exemple étend `Java` en lui fournissant des opérateurs de pattern-matching [142]. Nous avons choisi d'implémenter un langage dédié à la définition de mission indépendant ; nous en verrons le design aux chapitres 4 et 5, et l'implémentation au chapitre 6.

3.2 La vérification hors ligne des systèmes hybrides

De nombreux formalismes ont été développés ou étendus pour la vérification formelle des systèmes hybrides. Parmi les plus influents, la théorie des automates hybrides de Henzinger [113] les modélise par des modes (états) décrits par des équations différentielles et des transitions entre modes résultant d'événements (des gardes). Lors des transitions, des évolutions discrètes de l'état peuvent se produire (sauts). Nombre de travaux reviennent alors à cette notation pour décrire les systèmes hybrides. [133] utilise par exemple une logique de réécriture appliquée à des automates hybrides linéaires pour vérifier le problème classique des réservoirs. Dans leur implémentation en *Real-Time Maude*, le problème est généralisé à un nombre non prédéfini de réservoirs : il s'agit de maintenir un niveau d'eau suffisant dans des réservoirs fuyant à l'aide d'une unique lance à eau.

Les réseaux Petri sont une autre manière de représenter des systèmes par modèles présentant des transitions discrètes. Un réseau de Petri est un graphe comprenant des noeuds de deux sortes (des *places* et des *transitions*). Des arcs orientés relient ces noeuds, et un marquage distribuant des jetons sur les *places* traduit l'état du système. Plusieurs méthodes ont été envisagées pour étendre les réseaux de Petri à la modélisation de systèmes hybrides [64]. Il s'agit par exemple de coefficienter les variables par des booléens pour traduire les différentes phases. Pour une action de freinage à décélération constante B, on écrit alors l'évolution de la vitesse $\frac{dv}{dt}(t) = q * B$ où q est un booléen qui prendra notamment la valeur 0 quand le système est à l'arrêt. Ce type de transformations est courant pour représenter des formules logiques en inégalité linéaire (voire par exemple la programmation linéaire mixte ou les systèmes à logique mixte dynamique [49]). Une autre proposition est d'ajouter des conditions de durée, par exemple au niveau des transitions. Les

8. Nous les avons aussi appelés par ailleurs langages avancés car l'activité de programmation associée comprend par exemple la gestion de branches d'exécution

réseaux de Petri hybrides intègrent quant à eux de nouvelles places et transitions pour distinguer celles discrètes de celles continues. Enfin, une autre solution est de considérer le réseau de Petri comme un système supervisant les équations différentielles régissant l'évolution du système. Dans cette approche, la partie continue du système est présentée séparément de sa partie discrète.

`Event-B` est un langage servant à décrire des systèmes à événements discrets par modèles à état et à les analyser. Il se distingue par l'utilisation prépondérante du concept de raffinement hérité de la méthode `B`, la conception comprenant la preuve de la conformité de chaque raffinement. `Event-B` est également utilisé pour la vérification de systèmes hybrides, avec des tentatives initiales directes [185] avant de l'étendre en `hybrid Event-B` pour y ajouter la description de comportements continus au moyen d'équation différentielles ordinaires [43]. Le langage est notamment outillé par la plateforme *open-source* `RODIN`.

Les programmes hybrides sont des représentations alternatives aux automates hybrides. Si les automates semblent *a priori* plus adaptés pour certains traitements (*e.g. model-checking*), les programmes sont souvent plus maniables pour la composition. Les programmes hybrides ont été mis en avant par les travaux de Platzer [151], associés à la logique dynamique différentielle pour leur vérification. Outillée par son assistant de preuve `KeYmaera X`, cette technique a été utilisée pour vérifier des contrôleurs de systèmes cyber-physiques [154, 137, 56]. Elle dispose de plus d'une chaîne de compilation vérifiée depuis la spécification de modèles en logique dynamique différentielle au code exécutable en `CakeML` [55]. Les travaux présentés dans cet ouvrage utilisent la logique dynamique différentielle, nous aurons l'occasion de revenir sur ce formalisme au chapitre 7.

Parmi les approches plus purement mathématiques, évoquons la théorie de la viabilité qui s'intéresse aux équations différentielles multivaluées, ou *inclusion différentielle* [41, 40]. Une inclusion différentielle est donc de la forme $x'(t) \in F(x(t))$ et constitue, pour reprendre le vocabulaire d'Aubin, la « chance » du système, tandis que les contraintes de viabilité, mathématiquement la contrainte $\forall t > 0, x(t) \in K$, correspond à la « nécessité ». De ces termes transparaissent les questions auxquelles la théorie de la viabilité cherche à répondre dans sa partie contrôle : quelles sont les évolutions possibles (chance) qui maintiennent le système en vie (nécessité). Cela se traduit en deux propriétés, celle du noyau de viabilité, défini comme l'ensemble des conditions initiales pour lesquelles il existe une évolution satisfaisant les contraintes K, et celle du noyau d'invariance, l'ensemble des conditions initiales pour lesquelles toute évolution maintient K. La théorie de la viabilité est applicable à de nombreux domaines, au rang desquels nous retrouvons les systèmes économiques, biologiques, écologiques, sociaux, ou robotiques. La version originale s'intéressait aux systèmes continus ; une extension a été présentée pour couvrir les systèmes hybrides par l'introduction d'*impulsions*, c'est-à-dire de vitesses infinies permettant de modéliser des comportements discrets [42].

Relativement voisine, la théorie des fonctions à barrière de contrôle s'intéresse également à caractériser l'espace des états au moyen de fonctions discriminant les états selon leur sûreté [178, 37]. Ces fonctions peuvent être vues comme analogues aux fonctions de Liapounov pour la stabilité des solutions de systèmes différentiels. La caractérisation par fonction à barrière de contrôle a donc été appliquée à l'étude de la sûreté des systèmes hybrides. Elle permet par exemple de ne pas systématiquement expliciter l'ensemble des états sûrs, ce qui n'est d'ailleurs pas toujours faisable [124].

Autre théorie mathématique à avoir été régulièrement appliquée à l'analyse des systèmes hybrides, la théorie des jeux prend la forme des jeux hybrides différentiels. Ceux-ci insistent alors sur la modélisation dynamique de l'environnement : le jeu oppose alors un contrôleur chargé de diriger le système à un environnement maîtrisant ses turbulences [130, 114, 152]. Il s'agit souvent de généralisations multijoueurs de formalismes impliquant jusqu'alors un acteur.

3.3 La vérification de l'exécution

La vérification de l'exécution (*Runtime Verification*, *Runtime Monitoring*, *etc.*) s'est promue comme un domaine à part entière de l'informatique depuis deux décennies. Elle se définit comme la discipline qui traite de l'étude, du développement et de l'application des techniques de vérification qui permettent de vérifier si une exécution d'un système examiné satisfait ou viole une propriété de correction donnée [128]. La vérification de l'exécution concerne donc avant tout la détection d'invalidation d'une propriété, sans s'occuper d'une éventuelle action de *réparation*, c'est-à-dire sans influencer l'exécution. Elle recouvre [110] :

- la vérification de la conformité par rapport aux spécifications, pour laquelle une exécution est interrogée par rapport à des spécifications formelles
- l'analyse prédictive en ligne, inférant d'une exécution des propriétés d'un système
- la découverte de spécifications, permettant de (re)trouver des spécifications à partir de traces d'exécution
- la visualisation de traces d'exécution
- plus généralement, l'étude des traces d'exécution pour synthétiser quelque information intéressante
- la protection contre les défaillances

La vérification de la conformité par rapport aux spécifications s'est par exemple traduite par la programmation orientée surveillance (*Monitoring-Oriented Programming*) [66, 65]. En effet, partant du constat que la vérification formelle de programme n'était pas toujours possible, du moins sans une certaine lourdeur, les auteurs ont défendu l'idée de considérer la spécification et l'implémentation comme un système unique. La programmation orientée surveillance propose ainsi un complément léger aux méthodes formelles plus traditionnelles telles que la preuve de théorème. Son développement a opté pour la séparation de la programmation de la logique de spécification du fait de l'absence de cadre logique universel pour l'expression d'exigences. Cela permet de plus de combiner le langage d'implémentation et la logique de spécification selon les habitudes des acteurs respectifs. Plus concrètement, la programmation orientée surveillance repose sur l'ajout d'annotations au code d'implémentation. L'environnement de développement extrait alors les annotations écrites en logique qui correspondent aux exigences formelles et les convertit en code exécutable dans le langage d'implémentation. Ainsi, bien que la programmation orientée surveillance cherche à réunir implémentation et spécification, des cloisons demeurent : les exigences deviennent des contenus épisématiques du programme, s'inscrivant sur un plan de signification différent du signe langagier, et correspondent généralement à des acteurs différents (voir figure 3.2)

Ces méthodes sont à rapprocher des techniques de tolérance aux pannes (*Fault-Tolerant Design*) qui sont développées pour produire des logiciels résilients. Les deux principales méthodes de ce domaine sont les blocs de récupération (*recovery blocks*) et la programmation multiversion (*N-version programming*). Les blocs de récupération comprennent une liste ordonnée de programmes, produits indépendemment, et un test d'acceptation. Ils fonctionnent par essai-erreur : à l'entrée du bloc l'état du système est intégralement sauvegardé ; un programme est exécuté et s'il produit une erreur ou si son résultat ne satisfait pas le test d'acceptation, l'état précédemment sauvegardé est restauré et un programme alternatif répondant aux mêmes exigeances est exécuté. Cette procédure est répétée jusqu'à satisfaction du test d'acceptation ou épuisement des programmes, auquel cas le bloc renvoie une erreur. La probabilité que deux programmes indépendemment développés rencontrent la même erreur est supposée négligeable, et les blocs de récupération accroissent ainsi la tolérance aux pannes. La deuxième technique usuelle de tolérance aux pannes est la programmation multiversion. Plusieurs programmes sont écrits in-

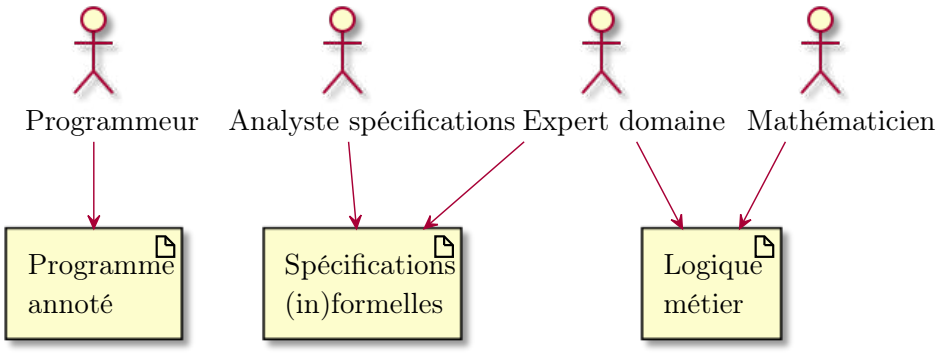


FIGURE 3.2 – Séparation des rôles dans la programmation orientée surveillance (adapté de [66])

dépendemment pour une même spécification. À l'exécution, tous les programmes sont exécutés et un mécanisme de vote permet de sélectionner la sortie que produira cette structure : ce mécanisme correspond par exemple à un vote de majorité. Ces deux techniques reposent ainsi sur une redondance fonctionnelle supposée assurer la robustesse du système. Trois catégories de redondances sont en effet à distinguer [174] :

- la réplication
- la redondance fonctionnelle qui produit des sorties identiques pour des entrées identiques
- la redondance analytique qui produit des sorties éventuellement différentes pour des entrées identiques, mais qui répondent à une même exigence

Un exemple de redondance analytique sont le volant et la pédale de frein d'une voiture pris dans le contexte de l'évitement d'obstacle : obliquer et freiner sont des réponses différentes permettant d'éviter un obstacle statique. L'architecture Simplex [173] met en oeuvre la redondance analytique pour intégrer des composants à hautes performances mais peu sûr dans des systèmes nécessitant un niveau de confiance supérieur. Pour cela, un mécanisme de vote peut à tout moment donner le contrôle à un composant moins performant mais offrant davantage de garanties. Cette architecture a été reprise dans les *Run-Time Assurance (RTA) units* [169], présentée dans la figure 3.3. La confiance accordée aux composants « sûrs » peut provenir de leur simplicité, de leur vérification (preuve formelle, tests), ou de leur certification. Le mécanisme de sélection doit notamment passer la main au composant robuste lorsque le système se trouve encore dans un état permettant un rétablissement. Pour cela, il est nécessaire de caractériser les états, par exemple au moyen de certificats de barrière (*Barrier Certificates*) [194]. Le composant qui doit être intégré présente en général des performances intéressantes sous quelque aspect : optimisation de la consommation d'énergie, planification de trajectoire ambitieuse, *etc.* Sa complexité le rend impropre à la vérification formelle (à l'instar de produits de l'apprentissage machine) ou il est démontré occasionnellement défaillant. En régime nominal, l'exécution est confiée à cette unité. Le mécanisme de surveillance et de sélection vérifie la satisfaction de propriétés sur l'état ou la consigne calculée. Si une faute est détectée, la main passe au composant de confiance, qui ramènera le système dans un état (plus) sûr.

La simplicité d'une structure de *RTA* la rend ubiquitaire. Elle est déclinée comme module de supervision de drone dans `SAFEGUARD` [88, 103]. Il s'agit d'une unité indépendante à embarquer sur un drone pour garantir un *geofencing* sur la navigation. Ce module dispose de son propre

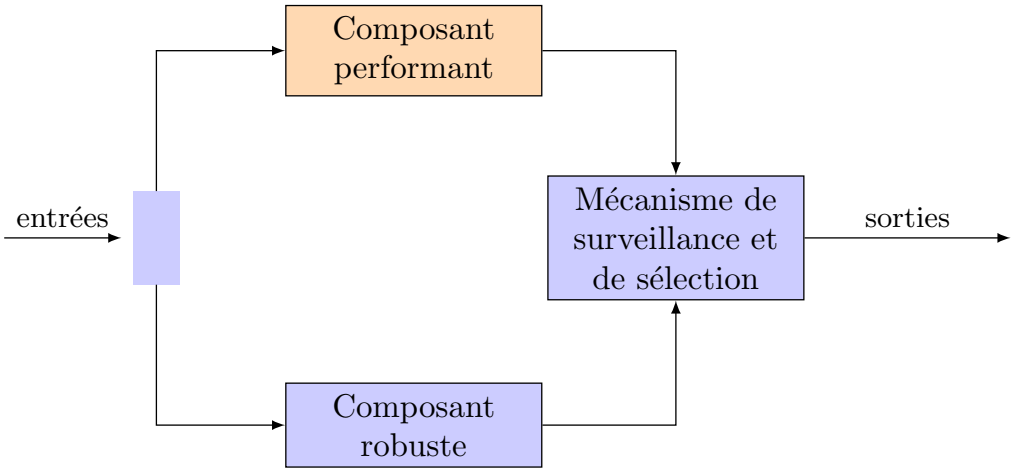


FIGURE 3.3 – *Run-Time Assurance unit.* Les éléments en bleu sont dignes de confiance tandis que le composant supposé plus performant est sujet au doute, sans que cela n'affecte la confiance globale en la structure.

système de localisation (GPS redondé d'un autre dispositif de localisation, ainsi que d'une horloge propre). Le module permet de filtrer les consignes envoyées aux drones de sorte à empêcher tout franchissement de *geofence* critique. Cependant, pour correctement superviser le drone, `SAFEGUARD` nécessite des composants de meilleure qualité que ceux embarqués sur le drone. Dans le cadre du projet Veridrone, un dispositif similaire est implémenté de manière purement logiciel sous la forme d'un *shim* [165]. Le *shim* intercepte le signal pour les moteurs d'un multicoptère à la sortie du module de mixage. Si la poussée proposée par le contrôleur n'est pas acceptée, le *shim* produit un signal correspondant à une poussée sûre. Des filtres similaires ont été pensés pour fonctionner également en mode piloté, en projetant la vitesse désirée dans l'espace des commandes sûres [107]. Dans le cadre de la programmation événementielle avec `Drona` (cf chapitre 2), une extension du langage a été proposée pour intégrer des composants non-vérifiés par des modules RTA programmables [81]. Enfin, évoquons ModelPlex [138] qui synthétise des moniteurs corrects par construction pour les systèmes cyber-physiques par preuve de théorème. Ces moniteurs, qui s'inscrivent dans une architecture Simplex, sont de trois catégories :

- des moniteurs de modèle, qui vérifient pour le modèle qu'il existe une transition entre l'état précédent et l'état courant du système (conformité du modèle)
- des moniteurs de contrôleur, qui vérifient la sortie d'un contrôleur concret par rapport à un modèle de contrôleur correct ; si la formule correspondante n'est pas satisfaite une commande d'un contrôleur secondaire doit être envoyée (partie proprement Simplex)
- des moniteurs prédictifs, qui évaluent la conséquence de déviations bornées du modèle afin de prédire si le prochain état pourrait ne plus être sûr

Les techniques de tolérance aux pannes du génie logiciel irriguent ainsi en amont le champs de la vérification de l'exécution et en aval celui dans lequel s'inscrit cet ouvrage, ce que traduiront les structures élémentaires de notre langage dédié `Sophrosyne`. Pour autant, gardons à l'esprit qu'il n'y a là aucune relation d'identité : les similitudes ne doivent pas occulter les différences, au rang desquelles nous retrouvons les objets d'études et la finalité. Pour les premiers, le logiciel n'est qu'une partie du système cyber-physique, et l'étude que nous menons ne peut le dissocier d'entrées et sorties dans un environnement non maîtrisé. Cela met à mal toute récupération d'un

état préalablement sauvegardé inhérente aux techniques de blocs de récupération. Une faute est un défaut, une imperfection, une faille, telle qu'un bug pour un logiciel, pouvant mener à une erreur, elle-même susceptible de provoquer une panne. Quant à la finalité, la tolérance aux pannes a pour objectif de produire un système qui continue d'exercer ses fonctions prévues en présence de fautes [95]. Si un système cyber-physique est naturellement sujet aux défaillances, ces considérations ne sont néanmoins pas au coeur de cet ouvrage ; nous nous intéresserons ici à du logiciel fonctionnant correctement, supposément en l'absence de fautes (dégradation physique du matériel, attaque malveillante), mais dans un environnement par essence incertain [9].

3.4 La vérification appliquée aux drones

Dans la littérature, la notion de mission recouvre alternativement une spécification haut-niveau – souvent une formulation d'objectifs –, ou une implémentation plus concrète. Cette distinction n'est dans le détail pas aussi manichéenne qu'elle ne le paraît, mais nous la conservons en discriminant selon que l'intérêt est porté sur la planification (production d'une mission à partir d'un plan) ou la mission elle-même. Dans le premier cas, les approches formelles sont utilisées pour définir rigoureusement des objectifs, puis pour dériver une mission idéalement optimale pour satisfaire ces objectifs. À notre connaissance, ces contributions communiquent avant tout sur la pertinence d'un formalisme, et se limitent généralement à des cas simples sur lesquels ils synthétisent des missions (faisables ou optimales). Pour des problèmes d'affectations de ressources (flotte de drones), on trouve ainsi des exemples formulés à base d'algèbres de processus [120], de modèles de Kripke [176] ou en Logique Temporelle Linéaire [115].

Des propositions ont été faites pour superviser l'exécution des drones avec `Lola`, pour la vérification de *geofences* [170] et plus généralement pour toute supervision liée à la sûreté de fonctionnement [33]. `Lola` [75] est un langage de spécification et de supervision en ligne et hors ligne basé sur des flux. De ce point de vue, le langage s'inscrit dans une lignée qui compte notamment `Lustre`, `Esterel`, ou `Signal`, desquels il se démarque par la possibilité de décrire des propriétés dont l'évaluation est différée. Dans [188], l'approche est intégrée aux concepts d'opération [10] pensée dans le cadre des scénarios spécifiques SORA (cf Chapitre 1) pour restreindre les états accessibles du drone.

`SAFEGUARD` permet de définir des *geofences* formulées comme des polygones d'inclusion ou d'exclusion. L'opérateur téléverse ces contraintes avec les paramètres de la dynamique de l'aéronef. Les algorithmes détectant la proximité et la violation des *geofences* ont été vérifiés avec l'assistant de preuve *Prototype Verification System*, à l'instar des autres algorithmes d'`ICAROUS`, une solution logicielle complémentaire apportant des fonctionnalités telles que la détection et l'évitement d'obstacles, la gestion de *geofences*, le suivi de cible, ou le retour à la mission [70]. Parmi les autres propositions de vérification formelle de *geofences*, [165] conduit ses vérifications au sein de l'assistant de preuve Coq. En partant de contrôleurs mono-dimensionnels, ils les composent jusqu'à obtenir un contrôleur conforme du *geofencing* 3D comprenant des géométries basiques telles que des couloirs cubiques ou parallélépipèdiques. Le contrôleur résultant a été testé et intégré au sein de l'autopilote Ardupilot.

9. L'incertitude comprend elle les défaillances de capteurs, ou les écarts entre les spécifications (modèles) et le comportement réel du système

10. Ce terme est généralement utilisé dans le contexte de la Défense « Document qui décrit la manière d'utiliser les forces, la chronologie retenue pour atteindre les objectifs fixés, et la façon dont il convient de synchroniser les différents moyens et ressources mis à disposition. » Journal officiel du 02/02/2008

3.5 Conclusion

Les modèles sont omniprésents en science. Nous avons rappelé certaines de leurs qualités, au premier rang desquelles l'efficacité et l'opérativité les rendent notablement attractifs. Cependant les modèles ne sont pas les systèmes qu'ils modélisent, et il convient de ne pas surinterpréter leurs vertus ; nous avons en effet averti au chapitre précédent des dangers d'étudier les systèmes cyber-physiques comme des formalismes interprétés. Du fait de leur nature infidèle, nous ne les considérerons pas dans une démarche explicative, ce qui justifie des modélisations occasionnellement laxistes, par exemple au moyen d'encadrements rudimentaires de l'état du système. La modélisation est toutefois utile pour la conception car un modèle s'analyse et éventuellement se prouve. De nombreux travaux se sont employés à vérifier les systèmes hybrides qui permettent de représenter les systèmes cyber-physiques. Une kyrielle de formalismes en résulte, et nous utiliserons pour nos travaux l'un d'entre eux, la logique dynamique différentielle, pour vérifier la correction théorique de nos missions (cf chapitres 7 et 8).

Conscients de l'inexactitude des modèles et de l'impossibilité de tout prouver, des méthodes de vérification en ligne ont été développées. Celles-ci reposent sur l'intégration de moniteurs pour superviser l'exécution ou vérifier la conformité du modèle et de l'exécution. Le langage que nous proposons, `Sophrosyne`, met en oeuvre la première catégorie de moniteurs : ils en constituent avec la programmation de mission le domaine d'application au sens de l'ingénierie dirigée par les modèles.

Deuxième partie

La chaîne outillée Sophrosyne

Chapitre 4

Sophrosyne : présentation générale

Ici le temps devient espace
Parsifal, Wagner

Il est temps d'introduire le langage qui supporte le travail exposé dans cet ouvrage. Il s'agit en effet de la clef de voute autour de laquelle s'étendent la vérification de mission (chapitre 8), l'exécution d'une mission (chapitre 6), et l'interface graphique (chapitre C). `Sophrosyne` est avant tout un langage dédié à la programmation et la vérification de missions pour systèmes cyber-physiques. Le présent chapitre en exposera la syntaxe concrète, accompagnée d'une sémantique informelle, et reprend en cela les développements de [191]. La grammaire de `Sophrosyne` fait l'objet de l'annexe A. La sémantique formelle fera l'objet du chapitre 5.

4.1 Considérations liminaires

Bien que `Sophrosyne` ait été pensé comme un langage dédié, il présente une grande expressivité. Le noyau du langage, conçu pour manipuler aisément le temps et pour définir des mécanismes de supervision, reste toutefois relativement petit. Nous allons tout d'abord introduire ces éléments : une bonne compréhension de ce socle restreint devrait grandement faciliter la lecture du reste du chapitre, et plus encore celle de la sémantique formelle (chapitre 5).

4.1.1 Les stratégies d'évaluation

Concernant le temps, `Sophrosyne` dispose d'entités immuables et d'entités temporalisées. Les premières n'évoluent qu'en cas d'instructions explicites dans le programme. Les secondes changent de valeur avec le temps. Ces différences se manifestent très concrètement dans le mode d'évaluation. Il existe différentes manières d'évaluer une expression, qui sont généralement séparées en méthodes strictes et méthodes non-strictes. Les méthodes strictes évaluent d'abord intégralement les arguments des expressions tandis que les méthodes non-strictes diffèrent ces évaluations. `Sophrosyne` utilise les deux modes d'évaluation, avec l'appel par valeur et l'appel par nom. Chacune de ces stratégies a ses intérêts lorsque l'on travaille avec des valeurs qui évoluent, comme la vitesse d'un drone. L'évaluation stricte nous servira à prendre un cliché instantané de cette vitesse, tandis que nous utiliserons l'évaluation différée pour exprimer des propriétés qui perdurent.

4.1.2 Les variables d'état

Les variables d'état constituent la première catégorie de valeurs temporalisées que nous rencontrons. Pour un modèle simpliste de drone, ces variables seront par exemple sa position, sa vitesse, et le temps physique. Elles font partie de l'interface entre le système cyber-physique réel et le système abstrait et sont en cela une contrainte exogène ; bien qu'il soit possible d'introduire d'autres variables, il n'est pas absurde de considérer en première approximation que ces variables représentent les données des capteurs du système cyber-physique réel. Par la suite, nous définirons par exemple une variable d'état pour la norme de la vitesse du drone. Dans le reste de l'ouvrage, nous utilisons la syntaxe concrète pour représenter les variables d'état. Celles-ci sont dès lors identifiables par le `@` préfixé, *e.g.* `@position`. Quand ces variables d'état représentent des objets complexes, nous accédons à un champ par la notation *pointée* : `@position.latitude` représente ainsi la latitude du drone.

4.1.3 Les affectations et les abstractions

Les variables d'état constituent une catégorie spéciale de variables : lors de l'exécution d'une mission, elles ne peuvent pas être modifiées par une affectation. Leur évolution sera en effet régie par l'action exécutée, soit directement lorsque nous étudierons symboliquement le système, soit indirectement lorsque leur valeur proviendra des capteurs du système réel ou simulé. `Sophrosyne` dispose de deux autres catégories de variables qui sont elles modifiables depuis le programme de mission, les variables affectées et les variables abstraites. Une affectation utilise l'appel par valeur comme stratégie d'évaluation ; elle permet donc de maintenir la valeur actuelle d'une variable. La syntaxe de cet opérateur est la suivante `a <- 2`. Les abstractions en revanche permettent de décrire des relations entre des variables dont les valeurs peuvent changer. La syntaxe de l'opérateur associée est `b -> 2 * a` pour définir une variable `b` dont la valeur sera toujours le double de celle de `a`. Nous insistons donc sur la différence entre

```
a <- 2 * @position.latitude
```

et

```
b -> 2 * @position.latitude
```

La première instruction enregistre la valeur actuelle de la latitude en l'associant à une variable `a`, tandis que la seconde instruction relie *dorénavant* la valeur de `b` au double de la latitude.

4.1.4 Le temps physique et le temps paramétrique

La gestion du temps est une composante importante de tout langage ou formalisme pour systèmes cyber-physiques. À l'instar de bien d'autres langages (Lustre [108], Esterel [51], $d\mathcal{L}$ [151]), nous adoptons l'hypothèse de synchronie. Nous distinguons dès lors les computations discrètes, supposées instantanées, de l'évolution continue, dont le domaine temporel est connexe. Cette dernière catégorie correspond aux constructions temporalisées dont nous avons esquissé le contour précédemment.

Les représentations paramétriques sont généralement propices à la description d'évolutions temporelles. Considérons un chemin linéaire dans le plan entre deux points A et B. Nous pouvons écrire ce chemin en utilisant une inégalité de convexité sur la position X :

$$\begin{cases} \vec{X}(\gamma) &= \gamma \times \vec{B} + (1-\gamma) \times \vec{A} \\ \gamma &\in [0,1] \end{cases} \tag{4.1}$$

Cela décrit un chemin et non pas une trajectoire : ni le temps physique ni la vitesse ne sont pour l'instant définis. Ce chemin pourrait être parcouru avec une vitesse constante $\vec{v_0}$, portée par $\overrightarrow{AB}$, auquel cas nous aurions la paramétrisation suivante du temps physique t (à l'origine du temps près) :

$$t(\gamma) = \frac{||\overrightarrow{AB}||}{||\vec{v_0}||}\gamma \tag{4.2}$$

Nous pouvons alors réécrire la position, cette fois comme fonction du temps physique

$$\begin{cases} \vec{X}(t) &= \vec{A} + t \times \vec{v_0} \\ t &\in \left[0, \frac{||\overrightarrow{AB}||}{||\vec{v_0}||}\right] \end{cases} \tag{4.3}$$

Par un développement similaire, en considérant à présent une trajectoire uniformément accélérée (avec pour accélération $\vec{a_0}$ et une vitesse initiale nulle), nous arrivons aux deux systèmes suivants :

$$\begin{cases} \vec{X}(\gamma) &= \gamma \times \vec{B} + (1-\gamma) \times \vec{A} \\ t(\gamma) &= \left(\frac{2\gamma||\overrightarrow{AB}||}{||\vec{a_0}||}\right)^{\frac{1}{2}} \\ \gamma &\in [0,1] \end{cases} \tag{4.4}$$

$$\begin{cases} \vec{X}(t) &= \vec{A} + \frac{1}{2}t^2 \times \vec{a_0} \\ t &\in \left[0, \left(\frac{2||\overrightarrow{AB}||}{||\vec{a_0}||}\right)^{\frac{1}{2}}\right] \end{cases} \tag{4.5}$$

Les systèmes (4.3) et (4.5) ont l'avantage d'être plus concis à l'écriture. Ils permettent également de répondre facilement à la question « Où se trouvera le système à tel moment ? ». En revanche, les systèmes ((4.1), (4.2)) et (4.4) font apparaître explicitement le fait que ces deux mouvements partagent un même chemin. Par ailleurs, ce découplage présente un intérêt quand nous nous intéressons à des comportements approximatifs, voire inconnus. Si nous ne disposons par exemple d'aucune information sur la vitesse, hormis qu'elle est positive vis-à-vis du trajet, l'équation de la position en fonction de γ reste valide, tandis que nous ne pouvons plus écrire de paramétrisation temporelle sans recourir à une écriture différentielle. Enfin, notons que le paramétrage en γ ne nous empêche en rien d'écrire des systèmes centrés sur le fait temporel : il nous suffira d'imposer $\gamma = t$ (ou par soucis de généralité sur l'origine du temps physique $t' = 1$).

Nous appelons γ le temps paramétrique, et nous le considérons comme une variable d'état. Par anticipation sur la présentation des actions en `Sophrosyne`, les propriétés qui régissent cette variable sont les suivantes :

- le temps paramétrique est borné et normalisé ($\gamma \in [0,1]$)
- quand une action commence, γ vaut 0
- quand une action se termine, γ vaut 1

En première approximation, γ peut être pensé comme un indicateur de progression dans l'action actuellement effectuée.

Reformulons notre example en `Sophrosyne` et considérons les points dans le plan cartésien (x, y), *e.g.* `A <- {x : 0, y : 1}` et `B <- {x : 10, y : -5}`. Le chemin entre `A` et `B` est ainsi l'abstraction suivante :

```
chemin -> {
  x : A.x * (1 - γ) + B.x * γ,
  y : A.y * (1 - γ) + B.y * γ
}
```

Et nous pouvons l'étendre à la trajectoire à vitesse `v` constante :

```
trajectoire -> {
  x : A.x * (1 - γ) + B.x * γ,
  y : A.y * (1 - γ) + B.y * γ,
  t : (B.x - A.x) * γ / v.x
}
```

Terminons enfin par remarquer que la séparation d'un *temps de l'exécution* et du *temps physique* n'est en rien une considération nouvelle ou propre à l'informatique. Il s'agit là d'un des principes fondamentaux de la musique, où l'on nomme *tempo* la vitesse de la pulsation servant d'étalon pour le rythme. Le tempo est généralement [11] le choix de l'interprète qui lui confère également une dynamique, en l'accélérant ou le ralentissant. La mesure – prise ici comme la spatialisation temporelle qu'opère l'écriture –, peut même s'affranchir entièrement de sa corrélation temporelle (elle en conserve toutefois la succession), de manière passagère avec des indications de dynamique telles que *ad libitum* ou *rubato*, ou sur un morceau à l'instar d'un récitatif. Ce relâchement ne prive pourtant pas l'exécution de structure, le rythme continuant de fournir une indication générale de proportion ou d'intention, et les musiciens jouant toujours *ensemble* (*viz.* en synchronie). Bien que ce développement ait été en partie la conséquence de l'absence d'outil de mesure précise du temps (avant l'invention du métronome au XIXème siècle, le pouls servait de référence), l'indétermination temporelle a perduré malgré leur apparition. Les caractéristiques du tempo que nous avons relevés se retrouvent dans notre approche du temps pour les systèmes cyber-physiques. L'introduction du temps paramétrique γ permet de même de spécifier des accélérations, des ralentissements, et des *ad libitum*, et une absence de contrainte sur le temps physique indique un rythme libre. Une partition de musique dispose d'une infinité d'interprétations possibles, que nous rapprochons de l'infinité d'exécution possible d'une même mission.

4.2 La définition de systèmes

Nous allons à présent nous intéresser à la définition de systèmes en `Sophrosyne`, c'est-à-dire à la modélisation du système cyber-physique. Cette définition n'a pas pour vocation d'expliquer le fonctionnement du système cyber-physique, mais elle en décrit les différentes évolutions escomptées. Elle est en cela une spécification du système en deux parties : *(i)* elle déclare l'interface, *i.e.* les primitives du système et *(ii)* elle décrit des propriétés portant sur ces éléments de l'interface. L'interface comprend les variables d'état, que nous avons rencontrées précédemment, et les actions que peut exécuter le système. Du point de vue de `Sophrosyne`, ces éléments d'interface sont des boîtes opaques. Les variables d'état sont des données en lecture seule : elles ne disposent d'aucun processus de mise à jour déterministe. De la même manière, nous n'*implémentons* pas d'actions en `Sophrosyne`. Quant aux propriétés, celles-ci nous sont utiles pour étudier les missions, *viz.* pour vérifier leur correction, ou pour estimer l'évolution de l'état du système cyber-physique.

4.2.1 La déclaration des variables d'état

Nous avons vu les caractéristiques d'une variable d'état dans la section 4.1.2. Il nous reste à présenter la syntaxe de leur déclaration. Celle-ci s'effectue en affectant une valeur initiale *e.g.*

11. ce paragraphe est écrit en ayant à l'esprit la musique classique occidentale. Il est certain que se référer à d'autres musiques apporterait un éclairage différent ; toutefois, si ces musiques sont susceptibles d'en prendre le contre-pied sur le détail, nous sommes convaincus que cela participe en définitive de la pertinence du propos, qui souligne simplement la richesse de la représentation temporelle dans le langage musical.

```
{
  @position : {x: 0, y: 0, z: 0},
  @velocity : {x: 0, y: 0, z: 0},
  @landed : True,
  @velocityNorm : 0,
  @t : 0
}
```

4.2.2 Les propriétés et l'invariant

L'évolution attendue des variables d'état est spécifiée par des *propriétés*. Une propriété est une expression booléenne qui restreint l'espace des états accessibles, par exemple `@position.x = 4`. Dans le cadre d'une propriété, nous pouvons également parler de la trajectoire que suit une variable d'état en la dérivant : `@position.x' > 1` Certaines propriétés représentent des contraintes physiques ou des relations fondamentales entre des variables d'état. Celles-ci contraignent constamment l'espace des états, et nous les définissons à part sous le terme d'*invariant* du système. Un ensemble d'invariant que nous pourions avoir pour les variables d'état précédemment définies est le suivant :

```
invariant : {
  @t' > 0,
  @position.x' = @velocity.x * @t',
  @position.y' = @velocity.y * @t',
  @position.z' = @velocity.z * @t',
  @landed  or @position.z > 0,
  not @landed or @position.z = 0,
  @velocityNorm ** 2 = @velocity.x ** 2 + @velocity.y ** 2 + @velocity.z **2,
  @velocityNorm >= 0
}
```

La première propriété nous permet de ne pas remonter le temps. Les trois suivantes corrèlent la position et la vitesse. Les quatrième et cinquième propriétés nous assurent que le drone est en l'air s'il n'est pas au sol et n'aura pas d'altitude négative, et enfin les dernières relient la vitesse à sa norme.

4.2.3 La définition des actions

Une action modélise une instruction primitive, mais de manière continue : elle décrit l'évolution des variables d'état en utilisant une propriété. L'exécution attendue d'une action satisfait la propriété du début à la fin de l'action. La bonne exécution d'une action peut toutefois nécessiter que le système soit dans un état adéquat : un drone devra avoir décollé avant de pouvoir rejoindre sa destination.

```
action goto(x, y, z):
    requires not @landed
    start <- @position
    @position.x = start.x * (1 - γ) + x * γ and
    @position.y = start.y * (1 - γ) + y * γ and
    @position.z = start.z * (1 - γ) + z * γ and
    @velocityNorm < 15;
```

Remarquons que l'action `goto` est sous-spécifiée : nous ne connaissons ni la vitesse, ni le temps. Du fait des propriétés invariantes introduites précédemment, nous pouvons toutefois les encadrer, par exemple pour le temps :

```
@t'**2 = ((x - start.x)**2 + (y - start.y)**2 + (z - start.z)**2) /
↪  @velocityNorm**2
```

en utilisant les propriétés liant vitesse, position et temps, celle définissant la norme de la vitesse par rapport à ses composantes, et la propriété régissant la position lors de cette action. De là, nous obtenons

```
@t' > ((x - start.x)**2 + (y - start.y)**2 + (z - start.z)**2)**(1/2) / 15
```

Cette inégalité s'interprète très simplement : le temps nécessaire à l'exécution de l'action est minoré par le rapport entre la distance à parcourir et la vitesse maximale du système.

Cette action impose tout de même le chemin : le drone rejoint sa destination en ligne droite. À l'extrême opposé, nous pouvons définir une action `goto2` ne faisant aucune hypothèse sur le chemin emprunté. Nous simplifions légèrement la notation en utilisant l'égalité par composante :

```
action goto2(X):  // X est ici un objet complexe (record)
    requires not @landed
    γ < 1 or
    γ = 1 and
    @position = X;
```

Ici les variables d'état évoluent librement pendant l'exécution de l'action (elles restent contraintes par l'invariant). En somme, la spécification de cette action dit que quand l'action se termine, le drone a rejoint sa destination.

4.3 La définition de missions

Nous disposons d'un système, nous allons maintenant pouvoir définir des missions pour ce système. La première étape est de déclarer le système qui devra exécuter cette mission : cela nous permettra d'accéder aux éléments de l'interface (variables d'état et actions) que nous avons précédemment modélisés. Ceci est fait par l'instruction `system`. Écrire une mission consiste à composer des actions du système, éventuellement en agrémentant cela de computations telles qu'un calcul de distance. La composition d'actions se fait séquentiellement, ou par l'utilisation de structures de supervision appelées *contextes*. `Sophrosyne` permet commodément de regrouper des portions d'exécution dans des sous-missions qui, déclaration de système exceptée, ressemblent en tout point aux missions ; en conséquence, ces structures sont également appelées *missions*. Une mission n'impliquant aucune action est une *fonction*.

4.3.1 Les contextes

Le branchement conditionnel est effectué grâce aux contextes en `Sophrosyne`. Ceux-ci sont des mécanismes de supervision, et contiennent trois éléments :

- une **garde**, c'est-à-dire une formule logique qui est continuellement évaluée
- une **mission principale**, qui est exécutée tant que la garde est satisfaite
- une **mission de repli**, qui est exécutée dès que la garde n'est plus satisfaite

La syntaxe correspondante est

```
with guard:
    main
else:
    fallback;
```

où `guard` est la garde, `main` la mission principale, et `fallback` la mission de repli.

Les contextes sont ainsi des dispositifs de contrôle pour adapter l'exécution selon la validité d'une contrainte : en particulier, dans le cas dégénéré où la mission principale et la mission de repli sont de simples computations n'affectant pas l'évaluation de la garde, un tel contexte se comporte comme une structure conditionnelle `if-then-else`. En cas d'une modification de valeur, la structure conditionnelle peut être interrompue. Dans l'exemple qui suit, la valeur finale de `a` est 70 (successivement $1 \times 2 \times 5 \times 7$) : lorsque x se voit affecté la valeur 1, un branchement s'opère.

```
x <- 0
a <- 1           // a = 1
with x = 0:
    a <- a * 2   // a = 2
    x <- 1
    a <- a * 3   // jamais atteint
else:
    a <- a * 5;  // a = 10
a <- a * 7       // a = 70
```

Le comportement par défaut d'un contexte, illustré dans la Figure 4.1, est dit *abortif* : le déclenchement de la mission de repli provoque l'abandon définitif de la mission principale. En effet, une fois la mission de repli complétée, l'exécution se poursuivra avec le bloc qui suit le contexte. `Sophrosyne` dispose d'une instruction changeant ce comportement, `resume`. Cette instruction, placée à la fin d'une mission de repli, rétablira la mission principale une fois la mission de repli exécutée. Plus précisément, la mission principale est reprise là où elle a été interrompue. Ces contextes sont dits *à reprise* (cf. Figure 4.2). Nous pouvons transformer le contexte de l'exemple précédent en un contexte à reprise en modifiant la mission de repli comme suit, donnant alors à `a` la valeur finale 210 (successivement $1 \times 2 \times 5 \times 3 \times 7$).

```
x <- 0
a <- 1           // a = 1
with x = 0:
    a <- a * 2   // a = 2
    x <- 1
    a <- a * 3   // a = 30
else:
    a <- a * 5   // a = 10
    resume;
a <- a * 7       // a = 210
```

Enfin, il existe une troisième catégorie de contextes qui seront traités dans le prochain paragraphe.

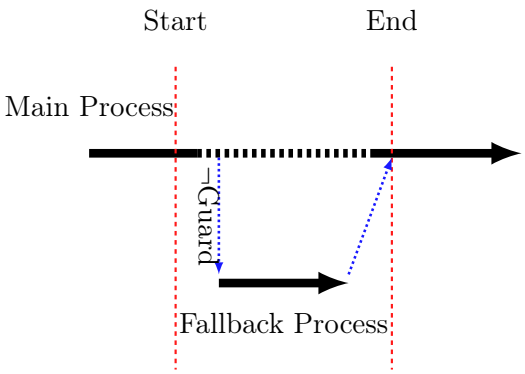


FIGURE 4.1 – Contexte abortif

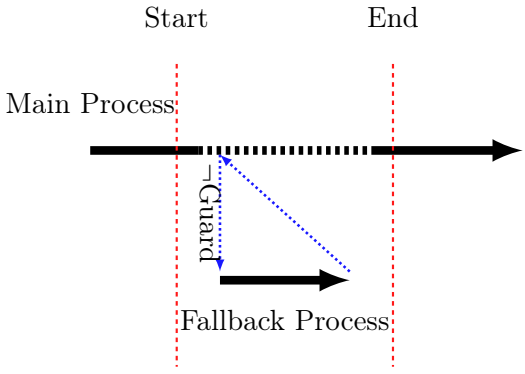


FIGURE 4.2 – Contexte à reprise

4.3.2 Les labels

Les contextes abortifs abandonnent la mission qui était exécutée, ceux à reprise la reprennent une fois la mission de repli exécutée, il nous reste donc une dernière catégorie de contextes à voir, ceux qui nous font reprendre depuis un point passé. La première étape est de définir un repère à partir duquel l'exécution reprendra. Cela est l'effet de l'instruction `label L` : elle définit un label, `L`, auquel il sera fait référence par un `resume L` pour revenir à ce point de la mission. Du fait des applications visées par le langage, une instruction de retour ne peut être introduite qu'au bout d'une mission de repli : le process de définition d'une mission en `Sophrosyne` comprend généralement plusieurs étapes, et construit d'abord la mission « idéale », linéaire, avant de lui rajouter les structures de supervision pour parer aux différents événements pouvant survenir lors de l'exécution de la mission par le système cyber-physique réel. Les systèmes qui nous intéressent effectuent pour l'essentiel des missions idéales ayant un début et une fin. En conséquence, il serait inadéquat de définir une boucle infinie dans le cadre de l'exécution principale de la mission. Ces questions de terminaison nous occuperont lorsque nous chercherons à vérifier une mission (cf. Chapitre 8), mais il reste possible d'introduire une boucle infinie en `Sophrosyne`, pour définir un asservissement par exemple. Concluons sur les labels en présentant une mission où le système cyber-physique doit parvenir à sa destination sans dépasser une vitesse donnée, sinon il lui faudra reprendre du départ. Il a le droit pour cela à trois essais, et finalement il émettra un son pour informer de son succès.

```
failures <- 0
label L
goto(0, 0, 5)
with failures < 3:
    with @velocityNorm < 5:
        goto(10, 10, 5)
    else:
        failures <- failures + 1
        resume L;
    happyBeep()
else:
    sadBeep();
```

4.4 Un exemple d'inspection par drone

Nous introduisons ci-dessous un modèle de système de drone que nous utiliserons comme fil rouge dans les chapitres suivants qui traitent du modèle analytique. On pourra à ce titre anticiper sur la suite et se référer à la partition de mission présentée figure 6.9 pour illustrer le propos. Pour garder une dimension acceptable au cadre de cet ouvrage et ne pas nous disperser en des écritures qui n'apporteraient rien de plus à notre propos, nous écartons un certain nombre de variables d'état pourtant usuelles pour la caractérisation de l'état d'un drone, à l'instar de l'attitude. L'étude se concentrera donc sur la cinématique, à savoir la position, la vitesse et l'accélération, et comprendra une gestion de ressource par la variable de charge de la batterie `@bat`.

Dans notre modèle, l'évolution du temps est affine par morceaux (voir ligne 12) : nous contraignons ainsi les relations entre position, vitesse et accélération. En particulier, une évolution cubique des variables de position lors d'une action suffit à garantir les profils respectivement quadratiques et linéaires des variables de vitesse et d'accélération (lignes 14 - 19).

Nous équipons notre système de quatre actions :

- `takeoff` (ligne 24) faisant décoller le drone jusqu'à l'altitude donnée
- `orbit` (ligne 32) pour décrire une trajectoire circulaire à altitude constante
- `goto` (ligne 40) pour rejoindre une destination (x, y, z)
- `land` (ligne 48) faisant atterrir le drone

La modélisation de ces actions traduit l'intention : dans cet exemple, notre attention est portée premièrement sur la position du drone. En ce cas, nous pourrions décrire le décollage par une simple interpolation linéaire : `@position.z' = alt`. Nous n'avons pas retenue cette solution, car l'introduction de l'accélération comme variable d'état invite à placer la vitesse dans l'espace des fonctions continues. L'approche proposée ici, quelque peu fantaisiste dans sa réalisation mais suffisante en guise d'illustration, consiste à faire croître puis décroître la vitesse de manière simple, par une fonction quadratique dont les racines sont 0 et 1. Comme la vitesse est proportionnelle à la dérivée de la position, nous pouvons modéliser cela aisément en associant la dérivée de la position au polynôme[12] $6 * \gamma * (1 - \gamma)$. On retrouve ce polynôme dans toutes les actions :

- pour l'altitude dans `takeoff` ligne 28
- pour la position angulaire dans le plan dans `orbit` lignes 34, 35
- pour la position dans `goto` lignes 43, 44, 45
- pour l'altitude dans `land` ligne 53

Nous introduisons une incertitude sur le temps nécessaire à la complétion des actions `takeoff`, `orbit`, et `land`. Du point de vue physique, celle-ci correspond à une variabilité la vitesse. Cependant, la faire porter explicitement par la vitesse nécessite de répercuter cela sur ses dérivées, en l'occurrence sur l'accélération. Il est donc plus efficace de l'exprimer au niveau de la dérivée du temps, ce que nous faisons aux lignes 29, 30, 37, 38, 54, 55. Pour l'action `goto`, nous n'incluons pas d'incertitude (voir ligne 46). Le trajet est fait avec une vitesse moyenne de 1 m/s, ce que nous pourrions écrire explicitement en calculant les produits scalaires selon chaque composante de la vitesse et en redistribuant selon un profil quadratique. Mais là encore, il est plus rapide d'écrire que la dérivée du temps est égale à la distance à parcourir.

Concernant la batterie, sa décharge est proportionnelle à l'accélération selon l'équation donnée ligne 20. Cette équation prend en compte la poussée nécessaire pour contrebalancer la gravité.

```
1 system UA:
```

12. Le coefficient dominant (6) sert à normaliser son intégrale entre 0 et 1.

```
  {
    @position : {x: 0, y: 0, z: 0},
    @velocity : {x: 0, y: 0, z: 0},
    @acceleration : {x: 0, y: 0, z: 0},
    @time : 0,
    @dtime : 1,
    @bat : 100
  }
  invariant: {
    @time' = @dtime,
    @dtime' = 0,
    @dtime > 0,
    @position.x' = @velocity.x * @dtime,
    @position.y' = @velocity.y * @dtime,
    @position.z' = @velocity.z * @dtime,
    @velocity.x' = @acceleration.x * @dtime,
    @velocity.y' = @acceleration.y * @dtime,
    @velocity.z' = @acceleration.z * @dtime,
    @bat' = -0.004 * (@acceleration.z + 9.89 + (@velocity.x**2 +
↪   @velocity.y**2)**0.5) * @dtime,
    @bat >= 0
  }

  action takeoff(alt):
    requires @position.z = 0
    @velocity.x = 0 and
    @velocity.y = 0 and
    @position.z' =  6 * alt * γ * (1 - γ) and
    @dtime >= 1 + 0.9 * alt and
    @dtime <= 1 + 1.1 * alt;

  action orbit(x, y, z, r):
    requires @x = x + r and @y = y and @z = z
    @position.x' = - @position.y * 12 * π * γ * (1 - γ) and
    @position.y' = @position.x * 12 * π * γ * (1 - γ) and
    @velocity.z = 0 and
    @dtime >= 0.9 * r and
    @dtime <= 1.1 * r;

  action goto(x, y, z):
    requires @position.z > 0
    start <- @position
    @position.x' = 6 * (1 - γ) * γ * (x - start.x) and
    @position.y' = 6 * (1 - γ) * γ * (y - start.y) and
    @position.z' = 6 * (1 - γ) * γ * (z - start.z) and
    @dtime * @dtime = (x - start.x)**2 + (y - start.y)**2 + (z - start.z)**2;
↪

```

```
48    action land:
49      requires @position.z > 0
50      startz <- @position.z
51      @velocity.x = 0 and
52      @velocity.y = 0 and
53      @position.z' =  -6 * startz * γ * (1 - γ) and
54      @dtime >= 1 + 0.9 * startz and
55      @dtime <= 1 + 1.1 * startz;;
```

À notre drone nous confions une mission d'inspection de ligne électrique. Pour cette mission, le drone vole à côté des lignes électriques, de pylône en pylône. Il effectue de plus un vol circulaire autour de chaque pylône. Si le niveau de charge de la batterie passe sous le seuil critique de 30%, le drone doit se diriger vers une zone d'atterrissage d'urgence située à proximité du deuxième pylône. Si pour une mission réelle nous ajouterions une détection de proximité du plus proche point d'atterrissage, nous nous contenterons de cette version simpliste pour notre exemple.

```
1  system UA
2  takeoff(10)
3  with @bat > 30:
4    goto(0, -5, 10)        // pylone 1
5    orbit(-8, -5, 10, 8)
6    goto (-3, 50, 22)      // pylone 2
7    orbit(-11, 50, 22, 8)
8    goto(15, 75, 13)       // pylone 3
9    orbit(7, 75, 13, 8)
10   goto(3, 50, 27)        // retour pylone 2
11   goto(0, 0, 15)         // retour a la base
12 else:
13   goto(@position.x, @position.y, 30)  // ascension pour evitement d'obstacles
14   goto(5, 45, 30);       // zone d'atterrissage d'urgence
15 land()
```

4.5 Conclusion

Notre contribution aux outils et méthodes de programmation des systèmes cyber-physiques mobiles se développe autour d'un langage dédié, `Sophrosyne`. Nous avons vu dans ce chapitre les idées principales de ce langage, en particulier les constituants d'un système, le paramètre d'avancement γ distinct du temps physique, et la composition de missions par contextes. Un exemple de couple système - mission a été introduit, et nous le retrouverons à différentes occasions dans la suite de l'ouvrage. Cette première présentation se voulait toutefois informelle, et, avant de détailler les ramifications de `Sophrosyne`, nous allons en expliciter une sémantique formelle au chapitre suivant.

Chapitre 5

Sophrosyne : la sémantique

Si les noms ne sont pas ajustés, le langage n'est pas adéquat.

Entretiens, Confucius

Le chapitre 4 a introduit de manière informelle le langage et ses structures. Cette présentation devrait être suffisante pour en saisir les idées directrices et permettre son utilisation. Cependant, ce gain d'accessibilité se fait inéluctablement au prix de la précision, si bien que toute tentative de formalisation de propriétés sur cette seule entrée en matière se verrait contrariée par les approximations qu'elle entretient. De même, cette indétermination risque de contrevenir à la conformité entre le modèle analytique et celui d'exécution. Ce chapitre remédie à cela en exposant une formalisation de la sémantique dénotationnelle par passage de continuation de `Sophrosyne`. Les causes contraires produisent ici les effets contraires, et nous espérons que la présentation précédente suffira à l'exégèse de ce chapitre. Nous conclurons par une mise en perspective de la supervision qu'introduit `Sophrosyne`. Le domaine d'application de `Sophrosyne` comme langage dédié transparaît en effet dans ses structures de contrôle de flux d'exécution, qui trouvent des précédents dans la littérature sur les langages de programmation.

5.1 Propos liminaires sur la formalisation de la sémantique

Ce chapitre présente une sémantique mathématique de `Sophrosyne`, c'est-à-dire une fonction d'interprétation associant un sens aux commandes du langage. Ce sens est donné par la transformation du magasin qui s'opère lors de l'exécution du programme. Comme le propose Strachey et Wadsworth [183], nous utilisons à cet effet une méthode à base de continuation pour pallier le problème des sauts d'exécution qu'introduit `Sophrosyne`. Dans ce cadre, la fonction d'interprétation *stmt* d'une commande se voit traditionnellement passer en plus de la commande l'environnement et la continuation :

$$stmt \in \text{Commandes} \to \text{Environnements} \to \text{Continuations}$$

Une continuation est un programme qui reste à exécuter, *i.e.* une transformation du magasin. L'effet d'une commande `cmd` avec κ pour continuation exécutée dans un environnement ρ se comprend donc comme la transformation d'un magasin σ en un nouveau magasin ς :

$$\varsigma = stmt(\texttt{cmd})(\rho)(\kappa)(\sigma)$$

Ceci peut se lire comme la transformation de σ en ς par l'interprétation de la commande `cmd` dans l'environnement ρ suivi de l'exécution du reste du programme κ :

$$\varsigma = \big(\kappa \circ stmt(\texttt{cmd})(\rho)\big)\sigma$$

La règle primordiale pour l'interprétation d'un programme est alors la composition séquentielle qui s'écrit :

$$stmt(\texttt{cmd}_0; \texttt{cmd}_1)(\rho)(\kappa) = stmt(\texttt{cmd}_0)(\rho)(stmt(\texttt{cmd}_1)(\rho)(\kappa))$$

Nous considérons cette règle acquise pour le restant du chapitre. Notons qu'une commande provocant un saut dans le programme effectuera une modification de la continuation, si bien que $\texttt{cmd}_1$ peut ne jamais être interprétée.

Les expressions nécessitent pour leur part une fonction d'interprétation légèrement différente, car elles renvoient une valeur en plus de potentiellement altérer le magasin. Cette valeur sera alors consommée par une continuation d'expression, une fonction acceptant une (ou plusieurs) valeur pour produire une continuation.

$$exp \in \text{Expressions} \to \text{Environnements} \to (\text{Valeurs} \to \text{Continuations}) \to \text{Continuations}$$

Pour différencier l'interprétation d'une expression de l'interprétation d'une valeur pure, nous renommerons la fonction d'interprétation *val*. Ainsi, $exp(1 + 4)$ correspond à l'interprétation du programme `"1+4"` tandis que $val(1 + 4)$ s'identifie à $val(5)$. Par ailleurs, nous opterons plus loin pour une autre manière de représenter ces transformations : la fragmentation de notre environnement en de mutliples sous-environnement rend cette première notation fastidieuse.

5.2 Les notations

Notre présentation de la sémantique formelle de `Sophrosyne` reposera sur les notations suivantes :

- $\emptyset$ désigne l'ensemble vide, `None`, la liste vide *etc.*
- $\rightsquigarrow$ désigne la composition séquentielle de deux continuations
- $\curvearrowright$ désigne le passage d'une valeur à une continuation
- $a :: b$ désigne les opérations de concaténation, d'insertion (ou de mise à jour dans le cas où b est une projection) de a dans b
- ϵ_i sera par convention une conjonction (ou énumération, le sens sera donné par la pragmatique) d'expressions. Il en va de même pour les identifiants ξ_i.
- _ : désigne un élément quelconque [13]

Nous représentons un couple d'une projection par un tuple associant l'argument à sa valeur, si bien que pour des éléments $a \in A$ et $b \in B$, l'élément (a, b) peut aussi bien appartenir à $A \times B$ qu'à $A \to B$, le choix étant dicté par le contexte. Toutefois, nous choisissons systématiquement le type $A \to B$ lorsque nous voulons garantir l'unicité vis-à-vis des éléments de A.

5.3 Les domaines sémantiques

Commençons par introduire les domaines sémantiques que nous utiliserons pour présenter la sémantique de `Sophrosyne`.

13. Il s'agit d'un élément auquel nous n'affectons pas d'identifiant car il sera en général détruit par la réécriture.

- Les valeurs, pouvant être des nombres ou des booléens $V \in \textbf{Valeurs} \equiv \mathbb{N} \cup \mathbb{R} \cup \mathbb{B}$
- Les identifiants $\xi \in \Xi$
- Les emplacements de magasin $\textbf{Emplacements} \subset \mathbb{N}$
- Les formules propositionnelles $\phi \in \Phi$
  Concernant les commandes du langage, nous réferons à l'annexe A pour en connaître la grammaire.

5.4 La configuration

5.4.1 La configuration générale

Une configuration est donnée par un tuple $(\kappa, \sigma, Loc_+, \rho, \rho_{state}, \mathcal{A}, \rho_\kappa, \rho_{action}, \Gamma, \mathcal{B}, \rho_{label}, \iota)$, où

- $\kappa \in \mathcal{K} \equiv (\Sigma \to \Sigma)$ est la **continuation** courante, le programme qu'il nous faut interpréter
- $\sigma \in \Sigma \equiv (\textbf{Emplacements} \to \textbf{Valeurs})$ est un **magasin**, projetant des emplacements sur des valeurs
- $Loc_+ \in \textbf{Emplacements}$ est le **bord du magasin**, indiquant le prochain emplacement libre
- $\rho \in \Xi \to \textbf{Emplacements}$ est un **environnement**, projetant des identifiants sur des emplacements ; il nous servira à stocker les variables affectées
- $\rho_{\text{state}} \in \Xi \to (\textbf{Emplacements} \times \textbf{Emplacements})$ est un **environnement d'état**, projetant des variables d'état vers des paires d'emplacements représentant la valeur actuelle de la variable d'état ainsi que la précédente (nous utiliserons ζ plutôt que ξ pour identifier les variables d'état)
- $\mathcal{A} \equiv [(\kappa, \rho)] \in (\mathcal{K} \times (\Xi \to \textbf{Emplacements}))^*$ est une pile de **continuations de fonctions** avec leur **environnement** ; elle nous sera utile pour naviguer entre le corps des fonctions et le fil d'exécution principal en gérant les environnements locaux
- $\rho_\kappa \in \Xi \to \mathcal{K}$ est un **environnement de continuation**, projetant des identifiants sur des continuations ; nous y mettrons les abstractions et les fonctions
- $\rho_{action} \in \Xi \to \mathcal{K}$ est un **environnement d'actions**, projetant des identifiants sur des continuations ; il sera peuplé des actions du système
- $\Gamma \equiv (\Gamma_\top, (\phi, \beta), \Gamma_\bot) \in \Gamma_{domain} \equiv (\Phi \times \mathcal{K})^* \times (\Phi \times \mathcal{K}) \times (\Phi \times \mathcal{K})^*$ est un *zipper* de **gardes** associées à leur mission de **repli** ; la pile de gauche correspond aux gardes valident, l'élément centrale à la garde que nous considérons, et la pile de droite aux gardes restant à valider. Les opérations afférentes sont détaillées dans le paragraphe 5.5.16
- $\mathcal{B} \equiv [(\alpha, \Gamma)] \in (\mathcal{K} \times \Gamma_{domain})^*$ est une pile de **continuations à reprendre** avec leurs **gardes** ; la reprise d'une mission nécessite de restaurer les contextes qui s'appliquaient
- $\rho_{label} \equiv \xi \to (\kappa, \Gamma) \in (\Xi \to (\mathcal{K} \times \Gamma_{domain}))$ est un **environnement de labels**, projetant des identifiants sur des continuations associées à leurs gardes
- $\iota \in \Phi$ est l'**invariant** du système

5.4.2 La configuration initiale

La configuration initiale est essentiellement vide, nous la présentons ci-dessous avec pour continuation initiale le programme `prog` à exécuter. L'état initial du magasin σ (et de son bord Loc_+) n'a pas d'effet sur la sémantique que nous verrons par la suite. La déclaration d'un système met à jour :

- les variables d'environnement ρ_{state}, en déclarant les variables d'états ainsi que leur valeur initiale

- par conséquence de ce qui précède, le magasin σ ainsi que son bord Loc_+
- l'invariant du système ι
- l'environnement d'actions ρ_{action}, en insérant les définitions d'actions

La mission affecte pour sa part :

- l'environnement ρ par les affectations (et en conséquence le magasin σ)
- l'environnement de continuations ρ_κ par la déclaration de fontions et les abstractions
- la pile de continuations de fonctions associées à leur environnement $\mathcal{A}$ pour gérer les appels de fonction, d'action, ...
- l'environnement de labels ρ_{label} quand un label est déclaré
- le *zipper* de gardes Γ, lorsqu'un nouveau contexte est exécuté
- la pile de continuations à reprendre $\mathcal{B}$ lors de l'évaluation des gardes

$$
\begin{aligned}
\kappa &\equiv && \texttt{prog} \\
\sigma &\equiv && \_ \\
Loc_+ &\equiv && \_ \\
\rho &\equiv && \emptyset \\
\rho_{\text{state}} &\equiv && \emptyset \\
\mathcal{A} &\equiv && \emptyset \\
\rho_\kappa &\equiv && \emptyset \\
\rho_{\text{action}} &\equiv && \emptyset \\
\Gamma &\equiv && (True, \_), \emptyset, \emptyset \\
\mathcal{B} &\equiv && \emptyset \\
\rho_{\text{label}} &\equiv && \emptyset \\
\iota &\equiv && \emptyset
\end{aligned}
$$

5.5 Les règles

Nous présentons l'interprétation sous forme de règles de réécriture :

$$\kappa \equiv \qquad \texttt{pass} \rightsquigarrow \varkappa \mapsto \qquad \varkappa$$

signifie que si la continuation actuelle κ correspond à une instruction interne fictive `pass` suivi d'une autre continuation $\varkappa$, elle peut être remplacée par $\varkappa$. Le formalisme que nous employons pour présenter ces règles de réécriture est un *pattern matching*. Pour être parfaitement explicite, la règle précédente s'écrit comme suit

$$\begin{array}{rcl}
\kappa \equiv & \texttt{pass} \rightsquigarrow \varkappa \mapsto & \varkappa \\
\sigma \equiv & \sigma \mapsto & \sigma \\
Loc_+ \equiv & Loc_+ \mapsto & Loc_+ \\
\rho \equiv & \rho \mapsto & \rho \\
\rho_{state} \equiv & \rho_{state} \mapsto & \rho_{state} \\
\mathcal{A} \equiv & \mathcal{A} \mapsto & \mathcal{A} \\
\rho_\kappa \equiv & \rho_\kappa \mapsto & \rho_\kappa \\
\rho_{action} \equiv & \rho_{action} \mapsto & \rho_{action} \\
\Gamma \equiv & \Gamma \mapsto & \Gamma \\
\mathcal{B} \equiv & \mathcal{B} \mapsto & \mathcal{B} \\
\rho_{label} \equiv & \rho_{label} \mapsto & \rho_{label} \\
\iota \equiv & \iota \mapsto & \iota
\end{array}$$

Cette règle d'évaluation de `pass`

1. ne dépend ainsi d'aucune composante de la configuration (excepté la continuation κ)
2. ne modifie aucune des composantes de la configuration (excepté la continuation κ)

De manière générale, quand une composante de la configuration présente ces deux caractéristiques (non pertinente pour le déclenchement de la règle et inchangée par la règle) nous l'omettons de l'écriture. Si seule la deuxième caractéristique s'applique, c'est-à-dire que la composante est inchangée, nous n'écrivons pas le membre de droite de la réécriture afin qu'il soit aisé de voir quelles composantes sont modifiées par la règle. Par ailleurs, afin d'alléger la notation des règles et sauf mention contraire, toutes les règles suivantes supposent toutes les gardes satisfaites, *id est*

$$\Gamma \equiv \Gamma_\top, \emptyset, \emptyset$$

5.5.1 L'évaluation d'une expression littérale

L'évaluation d'une valeur littérale est transparente

$$\begin{array}{rcl}
\kappa \equiv & exp(V) \curvearrowright \varkappa \mapsto & V \curvearrowright \varkappa
\end{array}$$

5.5.2 Le chargement d'une variable affectée

La valeur de la variable est directement cherchée dans le magasin.

$$\begin{array}{rcl}
\kappa \equiv & exp(\xi) \curvearrowright \varkappa \mapsto & V \curvearrowright \varkappa \\
\rho \equiv & (\xi, L) :: \varrho & \\
\sigma \equiv & (L, V) :: \varsigma &
\end{array}$$

5.5.3 Le chargement d'une variable abstraite

La variable est remplacée par la continuation correspondante κ_ξ.

$$\begin{aligned} \kappa \equiv & \quad exp(\xi) \curvearrowright \varkappa \mapsto & \kappa_\xi \curvearrowright \varkappa \\ \rho_\kappa \equiv & \quad (\xi,\, \kappa_\xi) :: \varrho_\kappa \end{aligned}$$

5.5.4 Le chargement d'une variable d'état

La valeur d'une variable est directement cherchée dans le magasin.

$$\begin{aligned} \kappa \equiv & \quad exp(\zeta) \curvearrowright \varkappa \mapsto & V \curvearrowright \varkappa \\ \rho_{\text{state}} \equiv & \quad (\zeta,\, \_,\, L) :: \varrho \\ \sigma \equiv & \quad (L,\, V) :: \varsigma \end{aligned}$$

5.5.5 Les opérations binaires

Nous donnons ici l'opération d'addition en guise d'illustration, les autres opérations habituelles sont définies de manière similaire (nous n'explicitons pas dans cet ouvrage la gestion des exceptions, dont l'intégration dans la sémantique apporterait ici une exhaustivité qui ne saurait contrebalancer la perte de lisibilité qu'elle produirait).

$$\kappa \equiv \quad exp(\epsilon + \epsilon') \curvearrowright \varkappa \mapsto \quad exp(\epsilon) \curvearrowright exp(\epsilon') \curvearrowright + \curvearrowright \varkappa$$

Avec la fonction intrinsèque

$$\kappa \equiv val(V) \curvearrowright val(V') \curvearrowright + \curvearrowright \varkappa \mapsto \quad val(V + V') \curvearrowright \varkappa$$

Pour les disjonctions, nous précisons l'ordre d'évaluation :

$$\kappa \equiv \quad exp(\epsilon \texttt{ or } \epsilon') \curvearrowright \varkappa \mapsto \quad exp(\epsilon) \curvearrowright \texttt{Cond}\big(val(True) \curvearrowright \varkappa,\, exp(\epsilon') \curvearrowright \varkappa\big)$$

où $\texttt{Cond} \in (\mathbf{Valeurs}, \mathbf{Valeurs}) \to \mathbb{B} \to \mathbf{Valeurs}$ est la fonction de sélection

$$\begin{cases} \top \curvearrowright \texttt{Cond}(a, b) = & a \\ \bot \curvearrowright \texttt{Cond}(a, b) = & b \end{cases}$$

De même pour les conjonctions :

$$\kappa \equiv \quad exp(\epsilon \texttt{ and } \epsilon') \curvearrowright \varkappa \mapsto \quad exp(\epsilon) \curvearrowright \texttt{Cond}\big(exp(\epsilon') \curvearrowright \varkappa,\, val(False) \curvearrowright \varkappa\big)$$

5.5.6 L'évaluation des systèmes différentiels

L'évaluation des systèmes différentiels est considérée comme un problème de satisfaisabilité sur des fonctions de *flux*. Ces fonctions de *flux* sont des fonctions dont le codomaine est l'ensemble des configurations Ω. L'évolution qui s'effectue en suivant un système différentiel est non déterministe ; il doit exister une fonction de *flux* Υ qui satisfait les contraintes d'évolution. Pour introduire ces contraintes, nous notons $\nu \in (\Omega \times \Xi) \mapsto$ Valeurs la fonction qui extrait d'une configuration la valeur actuelle d'un identifiant de variable d'état.

Les équations proviennent des invariants ainsi que des propriétés de l'action exécutée.

$$\begin{aligned}
\kappa \equiv \quad & stmt(\texttt{evolve}(\zeta_i' = \theta_i \ \texttt{and} \ \zeta_j = \epsilon_j)) \rightsquigarrow \varkappa \mapsto && \varkappa \\
\rho_{\text{state}} \equiv \quad & \{(\zeta_i, L_i^0, L_i^1)\} && \\
\sigma \equiv \quad & \{(L_i^0, _)\} :: \{(L_i^1, \zeta_{i-})\} :: \varsigma \mapsto && \{(L_i^0, \zeta_{i-})\} :: \{(L_i^1, \zeta_{i+})\} :: \varsigma
\end{aligned}$$

avec les ζ_{i+} qui satisfont la contrainte

$$\exists \Upsilon : [\gamma_-, \gamma_+] \to \Omega$$

$$\begin{cases}
\forall \zeta_i \ \lambda x \cdot \nu(\Upsilon(x), \zeta_i) \in \mathcal{C}^1[\gamma_-, \gamma_+] & \text{variables d'état définies et dérivables} \\
\quad \nu(\Upsilon(\gamma_-), \zeta_i) = \zeta_{i-} & \text{valeurs initiales des variables d'état} \\
\quad \nu(\Upsilon(\gamma_+), \zeta_i) = \zeta_{i+} & \text{nouvelles valeurs des variables d'état} \\
\forall x \in [\gamma_-, \gamma_+] \frac{\mathrm{d}\nu(\Upsilon(x), \zeta_i)}{\mathrm{d}x} = \theta_i & \text{satisfaction des contraintes différentielles} \\
\forall \zeta_j \ \nu(\Upsilon(x), \zeta_j) = \epsilon_j & \text{statisfaction des contraintes de domaine}
\end{cases}$$

Les systèmes différentiels peuvent également être composés par disjonction. Dans ce cas, l'évolution suit l'une des clauses de manière non déterministe.

Cette définition s'appuie sur l'interprétation des équations différentielles en logique dynamique différentielle [151]. Les systèmes différentiels que nous considérons comprennent également des équations implicites $(\zeta_j = \epsilon_j)$, que l'on peut, en première approximation, rapprocher du domaine des équations différentielles dans la logique dynamique différentielle. Une différence persiste : la transformation de ces équations implicites vers la logique dynamique différentielle fait également intervenir une affectation. Cette différence se manifeste également dans la sémantique par la fonction `initEvolve`. Dans le cadre d'une évolution, écrire `@x = @y` produit une affectation si les valeurs de `@x` et `@y` sont initialement différentes. La variable affectée est la première variable d'état apparaissant dans la clause : dans le cas précédent, il s'agira de `@x`. Les clauses pouvant être associées par conjonction, les valeurs affectées sont solutions d'un système d'équations. Par exemple, `@x = 2 * @y and @y = @z` affectera la valeur de `@z` à `@y` et la valeur de `2 * @z` à `@x`.

$$\begin{aligned}
& && \forall j \ val(\zeta_j) = exp(\epsilon_j) \\
\kappa \equiv \quad & stmt(\texttt{initEvolve}(\zeta_i' = \theta_i \ \texttt{and} \ \zeta_j = \epsilon_j)) \rightsquigarrow \varkappa \mapsto && \varkappa \\
\rho_{\text{state}} \equiv \quad & (\zeta_j, L_j^0, L_j^1) :: \varrho && \\
\sigma \equiv \quad & \{(L_j^0, _)\} :: \{(L_j^1, \zeta_{j-})\} :: \varsigma \mapsto && \{(L_j^0, \zeta_{j-})\} :: \{(L_i^1, \zeta_{j+})\} :: \varsigma
\end{aligned}$$

Enfin, remarquons qu'en l'absence de contrainte, c'est-à-dire en l'absence d'invariant et lorsque la propriété associée à l'action est une tautologie, l'évolution des variables d'états est complètement non-déterministe. Nous reviendrons sur cet aspect quand nous traiterons de l'exécution de `Sophrosyne` sur un système cyber-physique réel.

5.5.7 L'affectation

L'affectation d'une variable se fait en deux temps : d'abord le membre de droite de l'affectation est évaluée, puis l'opération intrinsèque d'affectation d'une valeur à un identifiant est exécutée. Enfin, l'affectation faite, les gardes sont réévaluées.

$$\begin{array}{rrrr} \kappa \equiv & stmt(\xi \leftarrow \epsilon) \rightsquigarrow \varkappa \mapsto & exp(\epsilon) \curvearrowright \texttt{assign}(\xi) \rightsquigarrow \Gamma_{\uparrow} \rightsquigarrow \varkappa \\ \rho_\kappa \equiv & (\xi,\ \_) :: \varrho_\kappa \mapsto & \varrho_\kappa \end{array}$$

Notons que dans le cas où l'identifiant était préalablement utilisé dans l'environnement des continuations ρ_κ (utilisation de la variable sous forme d'abstraction par exemple), nous supprimons cet enregistrement. L'opération d'affectation dans l'environnement est donnée comme suit :

$$\begin{array}{rrr} \kappa \equiv & val(V) \curvearrowright \texttt{assign}(\xi) \rightsquigarrow \varkappa \mapsto & val(V) \curvearrowright \texttt{assignLoc}(L) \rightsquigarrow \varkappa \\ \rho \equiv & (\xi, L) :: \varrho & \end{array}$$

Enfin, précisons l'opération d'affectation au magasin :

$$\begin{array}{rrr} \kappa \equiv & val(V) \curvearrowright \texttt{assignLoc}(L) \rightsquigarrow \varkappa \mapsto & \varkappa \\ \sigma \equiv & \sigma \mapsto & (L, V) :: \sigma \end{array}$$

5.5.8 L'abstraction

L'opération d'abstraction enregistre la continuation (le membre de droite) dans l'environnement de continuation. Cela fait, les gardes sont réévaluées.

$$\begin{array}{rrr} \kappa \equiv & stmt(\xi \rightarrow \epsilon) \rightsquigarrow \varkappa \mapsto & \Gamma_{\uparrow} \rightsquigarrow \varkappa \\ \rho_\kappa \equiv & \rho_\kappa \mapsto & (\xi,\ exp(\epsilon)) :: \rho_\kappa \\ \rho \equiv & (\xi,\ \_) :: \varrho \mapsto & \varrho \end{array}$$

De même que pour l'affectation, il peut être nécessaire de retirer l'identifiant de l'environnement ρ.

5.5.9 La définition d'un système

La définition d'un système enregistre la continuation permettant son initialisation.

$$\begin{array}{rrrr} \kappa \equiv & stmt(\texttt{system}\ \xi : & & \\ & \{\zeta_i : \epsilon_i\}\ \texttt{invariant} : \{\phi\}\ S;) \rightsquigarrow \varkappa \mapsto & & \varkappa \\ \rho_\kappa \equiv & \rho_\kappa \mapsto & (\xi,\ val(0) \curvearrowright \texttt{statevar}(\gamma) \rightsquigarrow exp(\epsilon_i) \curvearrowright \texttt{statevar}(\zeta_i) & \\ & & \rightsquigarrow \texttt{invariant}(\phi) \rightsquigarrow stmt(S)) :: \rho_\kappa & \end{array}$$

5.5.10 La déclaration du système

La déclaration du système invoque la continuation enregistrée dans l'environnement de continuations. De plus, elle remet à zéro l'environnement d'actions.

$$\begin{array}{rrr} \kappa \equiv & stmt(\texttt{system}\ \xi) \rightsquigarrow \varkappa \mapsto & \kappa_\xi \rightsquigarrow \varkappa \\ \rho_\kappa \equiv & (\xi,\, \kappa_\xi) :: \varrho_\kappa & \\ \rho_{\text{action}} \equiv & \_ \mapsto & \emptyset \end{array}$$

5.5.11 La déclaration des invariants

Les invariants d'un système sont enregistrés à part, dans ι.

$$\begin{array}{rrr} \kappa \equiv & \texttt{invariant}(\phi) \rightsquigarrow \varkappa \mapsto & \varkappa \\ \iota \equiv & \iota \mapsto & \phi\ \texttt{and}\ \iota \end{array}$$

5.5.12 La déclaration des variables d'états

La déclaration d'une variable d'état requiert l'allocation de deux emplacements dans le magasin. Le premier emplacement n'est pas affecté, car le système ne dispose pas encore d'historique de valeurs. La valeur de cet emplacement sera sur-écrite avant toute utilisation. Le second emplacement enregistre la valeur initiale de la variable d'état.

$$\begin{array}{rrr} \kappa \equiv & val(V) \curvearrowright \texttt{statevar}(\zeta) \rightsquigarrow \varkappa \mapsto & \varkappa \\ \sigma \equiv & \sigma \mapsto & (L + |V|, V) :: \sigma \\ \rho_{state} \equiv & \rho_{\text{state}} \mapsto & (\zeta, Loc, Loc + |V|) :: \rho_{\text{state}} \\ Loc_+ \equiv & Loc \mapsto & Loc + 2 \times |V| \end{array}$$

5.5.13 La définition d'une fonction ou d'une mission

La définition d'une fonction ou d'une mission prépare la projection des paramètres sur les arguments.

$$\begin{array}{rrr} \kappa \equiv & stmt(\texttt{def}\ \xi(\epsilon) :\ S;) \rightsquigarrow \varkappa \mapsto & \varkappa \\ \rho_\kappa \equiv & \rho_\kappa \mapsto & (\xi,\, \texttt{bind}(\epsilon) \rightsquigarrow stmt(S)) :: \rho_\kappa \end{array}$$

$$\begin{array}{rrr} \kappa \equiv & stmt(\texttt{mission}\ \xi(\epsilon)) :\ S; \rightsquigarrow \varkappa \mapsto & \varkappa \\ \rho_\kappa \equiv & \rho_\kappa \mapsto & (\xi,\, \texttt{bind}(\epsilon) \rightsquigarrow stmt(S)) :: \rho_\kappa \end{array}$$

5.5.14 La fonction `bind`

La fonction intrinsèque `bind` crée ou met à jour une paire identifiant - emplacement de l'environnement ainsi qu'une paire emplacement - valeur du magasin.

$$\begin{array}{rrr} \kappa \equiv & val(V) \curvearrowright \texttt{bind}(\xi) \rightsquigarrow \varkappa \mapsto & \varkappa \\ \rho \equiv & \rho \mapsto & (\xi, Loc) :: \rho \\ \sigma \equiv & \sigma \mapsto & (Loc, V) :: \sigma \\ Loc_+ \equiv & Loc \mapsto & Loc + |V| \end{array}$$

5.5.15 La mise à jour déterministe d'une variable d'état `set`

Cette fonction intrinsèque permettra de changer la valeur de γ lors des restaurations de continuation des contextes.

$$\begin{array}{rrr} \kappa \equiv & \texttt{set}(\xi,\, val(V)) \rightsquigarrow \varkappa \mapsto & \varkappa \\ \rho_{\text{state}} \equiv & (\xi,\, L^0,\, L^1) :: \varrho_{\text{state}} \mapsto & (\xi,\, L^1,\, L^0) :: \varrho_{\text{state}} \\ \sigma \equiv & (L^0,\, \_) :: \varsigma \mapsto & (L^0,\, V) :: \varsigma \end{array}$$

5.5.16 Les opérations sur Γ

Cette section introduit les opérations applicables au *zipper* des gardes

1. L'invalidation
   La liste de gauche du *zipper* représentant des gardes satisfaites, un mouvement vers la gauche invalide une garde γ_i.

   $$\begin{array}{rrr} \kappa \equiv & \Gamma_{\leftarrow} \rightsquigarrow \varkappa \mapsto & \varkappa \\ \Gamma \equiv & \gamma_i :: \Gamma_\top,\, \gamma_j,\, \Gamma_\bot \mapsto & \Gamma_\top,\, \gamma_i,\, \gamma_j :: \Gamma_\bot \end{array}$$

   où

   $$\gamma_i \equiv (\phi_i, \beta_i) \text{ si possible, sinon } \emptyset$$

2. La validation
   La liste de gauche du *zipper* représentant des gardes satisfaites, un mouvement vers la droite revient à considérer la garde actuelle γ_i comme satisfaite.

   $$\begin{array}{rrr} \kappa \equiv & \Gamma_{\rightarrow} \rightsquigarrow \varkappa \mapsto & \varkappa \\ \Gamma \equiv & \Gamma_\top,\, \gamma_i,\, \gamma_j :: \Gamma_\bot \mapsto & \gamma_i :: \Gamma_\top,\, \gamma_j,\, \Gamma_\bot \end{array}$$

   où

   $$\gamma_i \equiv (\phi_i, \beta_i) \text{ si possible, sinon } \emptyset$$

3. L'invalidation générale
   L'invalidation générale *rembobine* le *zipper*. Elle est définie récursivement à partir de l'opération d'invalidation.
   Terminaison

   $$\begin{array}{rrr} \kappa \equiv & \Gamma_{\uparrow} \rightsquigarrow \varkappa \mapsto & \varkappa \\ \Gamma \equiv & \emptyset,\, \emptyset,\, \Gamma_\bot & \end{array}$$

   Récursion

   $$\begin{array}{rrr} \kappa \equiv & \Gamma_{\uparrow} \rightsquigarrow \varkappa \mapsto & \Gamma_{\rightarrow} \rightsquigarrow \Gamma_{\uparrow} \rightsquigarrow \varkappa \\ \Gamma \equiv & \Gamma_\top,\, (\phi, \beta),\, \Gamma_\bot & \end{array}$$

4. La suppression
   La suppression s'effectue sur la dernière garde satisfaite.

$$\begin{array}{rrr} \kappa \equiv & \Gamma_{\_} \rightsquigarrow \varkappa \mapsto & \varkappa \\ \Gamma \equiv & (\phi, \beta) :: \Gamma_\top,\ \emptyset,\ \emptyset \mapsto & \Gamma_\top,\ \emptyset,\ \emptyset \end{array}$$

5.5.17 L'initialisation de la mission de repli

L'initialisation de la mission de repli remet γ à zéro et charge la continuation correspondante. Elle prépare également le retour de la mission principale. Pour cela, elle insère en tête de la continuation enregistrée la restauration de la valeur de γ ainsi qu'une réévaluation complète des gardes.

$$\begin{array}{rrr} \kappa \equiv & \texttt{fallback} \rightsquigarrow \varkappa \mapsto & \texttt{set}(\gamma,\ val(0)) \rightsquigarrow \beta \\ \Gamma \equiv & \Gamma_\top,\ (\phi, \beta),\ \Gamma_\bot \mapsto & (\phi, \beta) :: \Gamma_\top,\ \emptyset,\ \emptyset \\ \mathcal{B} \equiv & \mathcal{B} \mapsto & (\texttt{set}(\gamma, V) \rightsquigarrow \Gamma_\uparrow \rightsquigarrow \varkappa,\ \Gamma) :: \mathcal{B} \\ \sigma \equiv & (L,\ V) :: \varsigma & \\ \rho_{state} \equiv & (\gamma,\ \_,\ L) & \end{array}$$

5.5.18 L'évaluation des gardes

Lorsqu'une garde est considérée (indice courant du *zipper*), celle-ci est évaluée. Si elle est satisfaite, elle est marquée comme tel par l'opération de validation $\Gamma_\rightarrow$. Sinon, la mission de repli associée est déclenchée. Notons que bien que le *zipper* des gardes n'est pas intégralement validé, l'évaluation de l'expression ϕ et de la fonction de sélection se fait immédiatement.

$$\begin{array}{rrr} \kappa \equiv & \kappa \mapsto & exp(\phi) \curvearrowright \texttt{Cond}(\Gamma_\rightarrow \rightsquigarrow \kappa,\ \texttt{fallback} \rightsquigarrow \kappa) \\ \Gamma \equiv & \Gamma_\top,\ (\phi, \beta),\ \Gamma_\bot & \end{array}$$

5.5.19 La définition d'une action

La sémantique de la déclaration d'une action comporte des différences par rapport à ce que la syntaxe fait transparaître. Les prérequis ϕ semblent disparaître, mais nous les retrouverons lors de la vérification formelle de mission : ils n'ont à proprement parler aucun effet sur l'exécution. Les instructions S pouvant intervenir dans le corps d'une action sont des affectations et abstractions de variables qui interviennent dans les systèmes différentielles de l'action. Par ailleurs, une action renvoie une valeur de vérité, permettant d'informer la continuation parente de la complétion de l'action.

$$\begin{array}{rrr} \kappa \equiv & stmt(\texttt{action}\ \xi(\xi_{\text{param}}) : & \\ & \texttt{requires}\ \phi\ S\ \psi;) \rightsquigarrow \varkappa \mapsto & \varkappa \\ \rho_{\text{action}} \equiv & \rho_{\text{action}} \mapsto & (\xi,\ bind(\xi_{\text{param}}) \rightsquigarrow stmt(S) \rightsquigarrow \\ & & stmt(\texttt{initEvolve}(\psi\ \texttt{and}\ \iota)) \rightsquigarrow \\ & & stmt(\texttt{evolve}(\psi\ \texttt{and}\ \iota)) \rightsquigarrow \\ & & exp(\gamma = 1) \curvearrowright \texttt{return}) :: \rho_{action} \\ \iota \equiv & \iota & \end{array}$$

5.5.20 Le renvoi

L'expression renvoyée est d'abord évaluée puis sa valeur est passée à la fonction intrinsèque `return`.

$$\kappa \equiv \quad stmt(\texttt{return}\ \epsilon) \rightsquigarrow \_ \mapsto \quad exp(\epsilon) \curvearrowright return$$

La fonction intrinsèque `return` restaure l'environnement et la continuation depuis la pile

$$\begin{array}{rrr} \kappa \equiv & val(V) \curvearrowright return \mapsto & val(V) \curvearrowright \varkappa \\ \mathcal{A} \equiv & (\varrho, \varkappa) :: \mathcal{A}' \mapsto & \mathcal{A}' \\ \rho \equiv & \_ \mapsto & \varrho \end{array}$$

5.5.21 La fonction `apply`

Une application empile l'environnement et la continuation actuels. Les valeurs passées en arguments seront consommées par la fonction `bind` (cf section 5.5.14). Cette première règle concerne l'application d'une action.

$$\begin{array}{rrr} \kappa \equiv & val(V) \curvearrowright \texttt{apply}(\xi) \curvearrowright \varkappa \mapsto & val(V) \curvearrowright \varkappa_\xi \\ \mathcal{A} \equiv & \mathcal{A} \mapsto & (\varkappa,\ \rho) :: \mathcal{A} \\ \rho \equiv & \rho & \\ \rho_{action} \equiv & (\xi, \varkappa_\xi) :: \varrho_{action} & \end{array}$$

Similairement, pour les fonctions et les missions :

$$\begin{array}{rrr} \kappa \equiv & val(V) \curvearrowright apply(\xi) \curvearrowright \varkappa \mapsto & val(V) \curvearrowright \varkappa_\xi \\ \mathcal{A} \equiv & \mathcal{A} \mapsto & (\varkappa,\ \rho) :: \mathcal{A} \\ \rho \equiv & \rho & \\ \rho_\kappa \equiv & (\xi, \varkappa_\xi) :: \varrho_\kappa & \end{array}$$

5.5.22 L'appel de fonction et de mission

Un appel de fonction évalue les arguments avant de les passer à l'application

$$\begin{array}{rrr} \kappa \equiv & exp(\xi(\epsilon)) \curvearrowright \varkappa \mapsto & exp(\epsilon) \curvearrowright \texttt{apply}(\xi) \curvearrowright \varkappa \\ \rho_\kappa \equiv & (\xi, \_) :: \varrho_\kappa & \end{array}$$

5.5.23 L'appel d'action

Un appel d'action évalue les arguments avant de les passer à la continuation de l'action. L'application renvoie une valeur de vérité représentant la complétion de l'action (cf 5.5.19). Si l'action est terminée, γ est remis à zéro en prévision pour la prochaine action et les gardes sont invalidées pour être réévaluées. Sinon, les gardes sont invalidées pour réévaluation et l'exécution boucle en reprenant la même continuation.

$$\begin{array}{rlll} \kappa \equiv & stmt(\xi(\epsilon)) \rightsquigarrow \varkappa \mapsto & exp(\epsilon) \curvearrowright \texttt{apply}(\xi) \curvearrowright \texttt{Cond}(\texttt{set}(\gamma, val(0)) \rightsquigarrow \Gamma_\uparrow \rightsquigarrow \varkappa, & \\ & & \Gamma_\uparrow \rightsquigarrow \kappa) & \\ \rho_{action} \equiv & (\xi,\ \_) :: \varrho_{action} & & \end{array}$$

5.5.24 Les contextes

Un contexte s'évalue en ajoutant la garde et la mission de repli associée au *zipper* de gardes pour vérification. Cette garde sera supprimée en arrivant au terme de la mission principale ou de celle de repli.

$$\begin{array}{rrr} \kappa \equiv & \mathit{stmt}(\texttt{with}\ \phi : \alpha\ \texttt{else}\ \beta;) \rightsquigarrow \varkappa \mapsto & \alpha \rightsquigarrow \Gamma_{\_} \rightsquigarrow \varkappa \\ \Gamma \equiv & \Gamma_\top, \emptyset, \emptyset \mapsto & \Gamma_\top, (\phi, \beta \rightsquigarrow \Gamma_{\_}), \emptyset \end{array}$$

5.5.25 La reprise `resume`

Lors d'une reprise, la continuation actuelle est oubliée, tandis que celle enregistrée sur la pile de continuation à reprendre est restaurée.

$$\begin{array}{rrr} \kappa \equiv & \mathit{stmt}(\texttt{resume}) \rightsquigarrow \_ \mapsto & \varkappa \\ \Gamma \equiv & \_ \mapsto & \Gamma' \\ \mathcal{B} \equiv & (\varkappa, \Gamma') :: \mathcal{B}' \mapsto & \mathcal{B}' \end{array}$$

5.5.26 La déclaration d'un label

Un label enregistre la continuation actuelle dans l'environnement de labels. L'environnement de gardes Γ est également enregistré, et un invalidation générale $\Gamma_\uparrow$ déclenchera la réévaluation de chaque garde lors de la restauration de la continuation.

$$\begin{array}{rrr} \kappa \equiv & \mathit{stmt}(\texttt{label}\ \xi) \rightsquigarrow \varkappa \mapsto & \varkappa \\ \Gamma \equiv & \Gamma_\top, \emptyset, \emptyset & \\ \rho_{\text{label}} \equiv & \rho_{\text{label}} \mapsto & (\xi \to (\Gamma_\uparrow \rightsquigarrow \varkappa, \Gamma)) :: \rho_{\text{label}} \end{array}$$

5.5.27 La reprise depuis un label

La continuation associée au label dans l'environnement de continuations est restaurée, ainsi que le *zipper* de gardes qui avait été sauvegardé. La continuation actuelle est abandonnée.

$$\begin{array}{rrr} \kappa \equiv & \mathit{stmt}(\texttt{resume}\ \xi) \rightsquigarrow \_ \mapsto & \varkappa_\xi \\ \Gamma \equiv & \_ \mapsto & \Gamma_\xi \\ \rho_{\text{label}} \equiv & \xi, \varkappa_\xi, \Gamma_\xi :: \rho_{\text{label}} & \end{array}$$

5.6 Structures de contrôle de flux analogues

La sémantique ainsi présentée, il nous semble utile de nous arrêter brièvement sur les trois structures de contrôle de flux d'exécution dont dispose `Sophrosyne` et de les mettre en perspective. Depuis la lettre de Dijkstra [87], les concepteurs de langages de programmation ont généralement cherché à s'éloigner des structures de contrôle proche du `GoTo`. La critique principale que faisait Dijkstra à l'égard de ces opérateurs de bas niveau était la difficulté qu'éprouvait le développeur pour comprendre rapidement l'état du programme, la spatialité du code étant décorrélée de la chronologie de l'exécution. Dans le cas qui nous occupe, cet argument nous

semble perdre de sa pertinence : indépendemment de ces structures, l'exécution étant en partie non-déterministe, nous sommes bien en peine d'identifier à tout instant l'état du système sans utiliser de ressources extérieures au code source.

Des opérateurs de contrôle similaires ont été proposés dans le langage `Gedanken` avec les instructions `label` et `GOTO` [164]. Une valeur de label est créée lorsque l'exécution atteint une instruction de labélisation. Les labels servent alors d'arguments pour les fonctions `GOTO`. Ces fonctions tranfèrent le contrôle de l'exécution à l'état computationnel qui a été enregistré avec le label. Pendant l'exécution, l'état de l'interpréteur abstrait est en effet donné par :

- un *contrôle*, listant les instructions du bloc actuel qui doivent encore être exécutées
- un *environnement*, liant les identifiants du bloc actuel
- un *dump*, c'est-à-dire une pile contenant les computations à effectuer dès lors que le bloc actuel est entièrement exécuté,
- une *memoire*, enregistrant les projections des références.

Ces éléments présentés, nous pouvons préciser les opérateurs sus-mentionnés. Un `label` enregistre le *contrôle* actuel, l'*environnement*, et le *dump*. L'application d'un `GOTO` avec ce label pour argument restaure le contrôle, l'environnement et le *dump*. De même que dans `Sophrosyne`, la mémoire est ici conservée. Dans `Gedanken`, Reynolds a opté pour faire des valeurs de label des entités de première classe, ce qui constitue la principale différence entre ces structures et le couple `label L` - `resume L` de `Sophrosyne`.

Il est souvent intéressant de considérer `Scheme` lorsque l'on analyse des opérateurs de contrôle de flux. Doté de la fonction `call/cc` (*call-with-current-continuation*), `Scheme` promeut les continuations comme entités de première classe (d'autres langages ont repris cette fonction, à l'instar de `Racket` ou `Ruby`). `call/cc` permet d'exprimer les opérateurs de contrôle traditionnels tels que les mécanismes d'exceptions (`try-throw-except`), le *threading*, les instructions d'échappements (`break`), les générateurs, *etc.* Notons que `call/cc` n'enregistre que la pile d'appel. Bien qu'utile au développeur pour définir les structures de contrôle adéquat, `call/cc` demeure toutefois bas niveau et une kyrielle de structures de contrôle ont été proposées en guise de remplacement ou de complément, souvent sous forme de dialecte de Scheme à l'instar des continuations délimitées, `prompt-control` [177], `reset-shift` [76] (nous renvoyons vers [159] pour une analyse générale de ces structures), des `engines` qui quantifient l'exécution [111], *etc.* Les structures `try-except` (nous utilisons ici la syntaxe `python`) sont également similaires, mais modifier au sein du bloc `try` une variable qui n'est pas directement liée à l'exception reste généralement considéré comme douteux. Plus anciennement, le langage `C` offre les instructions `setjmp-longjmp` pour sauvegarder et restaurer les registres et l'adresse de retour. Elles sont en cela à rapprocher de `call/cc`. Ce type de structure de contrôle peut être utilisé pour la programmation par retour sur trace (voir par exemple la structure `setChoicePoint-fail` qui étend `setjmp-longjmp` [141], ou les `worlds` de `CLAIRE` [63]).

5.7 Conclusion

Ce chapitre a ainsi introduit la sémantique formelle de `Sophrosyne`, qui est un préalable à l'élaboration de toute théorie de la correction de ses missions. Nous avons vu que les structures de contrôle de flux sur lesquelles `Sophrosyne` repose s'inscrivent dans une longue histoire de l'utilisation des continuations. L'originalité de notre démarche est toutefois que ce sont les seules structures que nous introduisons (cela suffit à rendre le langage Turing complet, cf théorème de la programmation structurée). Un autre aspect notable est son application aux systèmes cyber-physiques : nous y retrouvons des traces de la caractérisation que nous avions faite au chapitre

2, par exemple l'introduction de variables en lecture seule pour représenter l'état ou le soucis de contrôler l'exécution à des fins de sûreté de fonctionnement avec nos contextes.

Chapitre 6

Sophrosyne : l'environnement logiciel

Dans la vie, nous combinons un plan ; mais celui-ci reste subordonné à ce qu'il plaira de faire au sort.

Aphorismes sur la sagesse dans la vie,
Schopenhauer

Dans les chapitres précédents nous avons présenté le langage `Sophrosyne` par la méthode : les concepts permettant d'écrire une mission ont été introduits, et nous verrons plus loin la deuxième partie portant sur la correction théorique et la vérification de ces missions. Avant cela, ce chapitre expose les principaux outils afférents, les grandes idées de l'architecture ayant été énoncées dans [192]. L'écosystème logiciel autour de `Sophrosyne` comprend en l'état les briques suivantes :

1. `sophrosyne`, le noyau qui permet l'analyse langagière d'un programme sophrosyne ainsi que les diverses conversions
2. `pysophrosyne`, la bibliothèque d'exécution
3. `symsophrosyne`, la bibliothèque de raisonnement sur les modèles analytiques
4. `sophrosyne-gui`, l'interface graphique sur laquelle nous reviendrons au chapitre 9

Le développement qui suit s'intéresse successivement au trois premières, qui ont chacune été regroupées en une bibliothèque `Python`.

6.1 Le noyau `sophrosyne`

La première fonction de la bibliothèque `sophrosyne` est d'assurer le traitement langagier. `Sophrosyne` est tout d'abord un langage dédié, dont la bibliothèque `sophrosyne` implémente le processus de compilation. On retrouve en son sein :

- un **analyseur lexical**
- un **analyseur syntaxique**
- un **analyseur sémantique**
- un **générateur de code source**

Cette section présente ces différents composants. Il ne s'agit en aucun cas de détailler un processus de compilation qui ne se veut pas original, mais seulement d'en tracer les linéaments des traitements effectués, des outils tiers utilisés, et des applications dérivées.

6.1.1 L'analyse lexicale

L'analyseur lexical transforme une chaîne de caractères en une séquence de lexèmes. Les lexèmes se divisent en trois catégories :

- les mots-clés et opérateurs
- les valeurs littérales
- les identificateurs

L'analyseur lexical est concrètement un automate fini non déterministe, dont la construction peut être confiée à un générateur d'analyseur syntaxique tel que `lex`. Nous avons ainsi utilisé la bibliothèque `SLY` [47] de `Python`, qui fournit un générateur d'analyseur syntaxique inspiré de `lex`. Parmi les identificateurs dont dispose `Sophrosyne`, on note la présence de variables d'états reconnaissables à leur préfixe `@`. La valeur littérale γ est quand à elle traduite par `_`.

L'analyse lexicale permet de plus de dériver divers outils utiles à l'écriture de programmes. Pour `Sophrosyne`, nous avons ainsi implémenté

- un module de coloration syntaxique avec `pygments` [60], une bibliothèque de coloration syntaxique applicable dans de nombreuses situations (*e.g.* LaTeX, HTML)
- un mode `emacs`, assurant la coloration syntaxique et l'indentation automatique pour l'écriture de programme `Sophrosyne`

6.1.2 L'analyse syntaxique

La deuxième étape de la compilation transforme la séquence de lexèmes en un arbre syntaxique abstrait, une représentation structurée du programme. Ici encore, l'écriture d'un analyseur syntaxique peut être confiée à un générateur d'analyseur syntaxique tel que `Yacc` ou `GNU Bison`. Pour `sophrosyne`, la bibliothèque `SLY` comporte également un tel générateur qui implémente le même algorithme d'analyse déterministe LALR(1) que `Yacc`.

La figure 6.1 illustre le passage d'un programme de mission `Sophrosyne` à son arbre syntaxique abstrait. Les transformations subséquentes s'opèrent aisément sur l'arbre syntaxique abstrait en mettant en oeuvre un pattern *Visitor* [102], et l'on construit ainsi :

- une impression élégante (*pretty-print*) de l'arbre syntaxique abstrait
- des générateurs de code (voir paragraphe 6.1.4)
- un analyseur sémantique

6.1.3 L'analyse sémantique

L'analyse sémantique que nous effectuons pour `Sophrosyne` comprend la résolution des noms et une vérification de type partielle. L'étape de résolution des noms :

- vérifie que les noms appelés sont préalablement définis
- annote les appels avec la catégorie de l'appelant (fonction, action ou mission)
- annote les chargements de variables selon leur mode d'évaluation (affectation ou abstraction)

L'annotation de l'arbre présenté précédemment (figure 6.1) est illustrée par la figure 6.2. L'analyse statique de typage que nous effectuons vérifie que la modification d'un attribut d'un objet complexe respecte le mode d'évaluation de l'objet. Ainsi, le code suivant produira une erreur

```
offset -> {x: @position.x, y: @position.y + 2}
offset.x <- 4
```

```
system UA
alt <- 5
with @battery > 0.3:
    takeoff(alt)
    inspectLocal()
else:
    land();
```

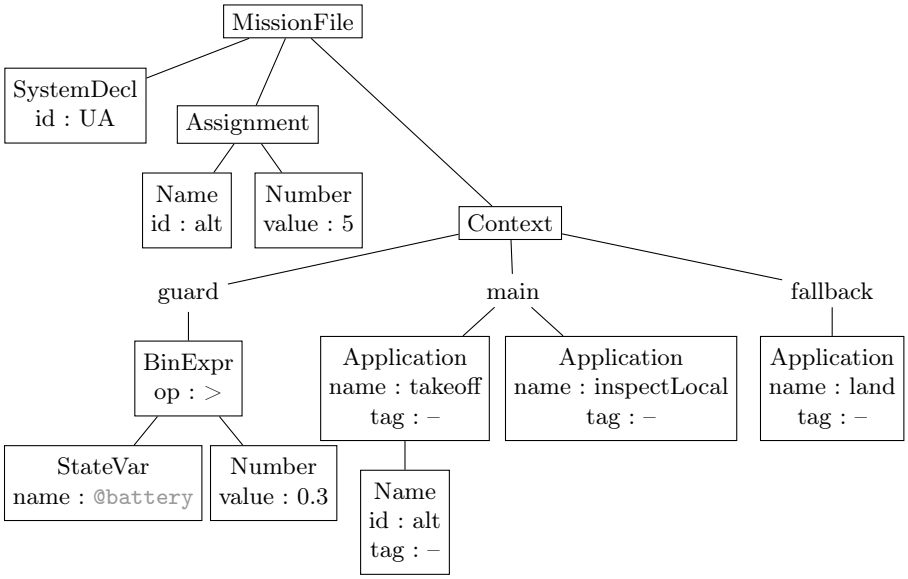


FIGURE 6.1 – Programme sophrosyne et l'arbre syntaxique dérivé

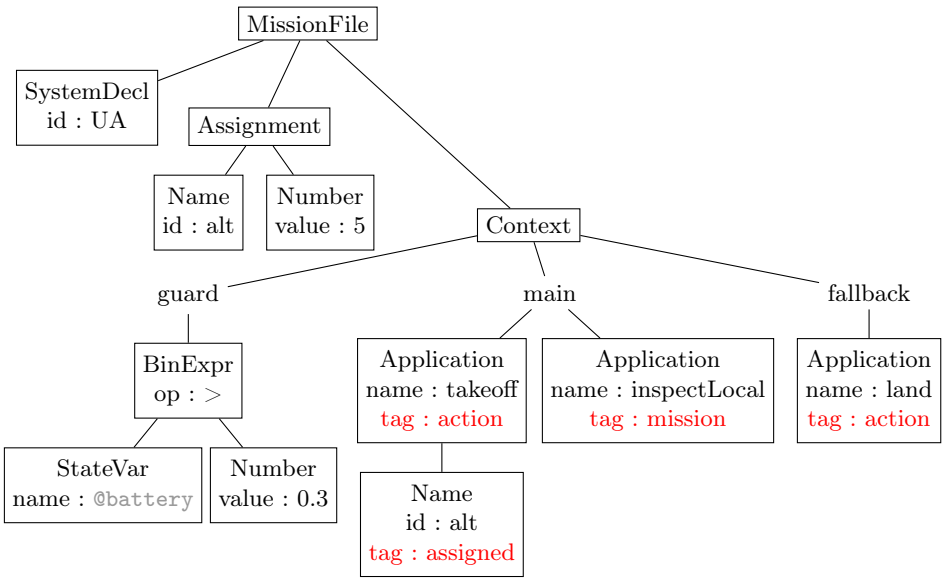


FIGURE 6.2 – Après analyse sémantique (voir figure 6.1)

6.1.4 La génération de code

La deuxième fonction de la bibliothèque `sophrosyne` est d'assurer les principales traductions :

- la génération de code exécutable (cf `pysophrosyne`, paragraphe 6.2)
- la génération de systèmes symboliques et d'obligations de preuve (cf `symsophrosyne`, paragraphe 6.3)
- la conversion entre les représentations `json` et `sophrosyne` des missions (cf `sophrosyne-gui`, paragraphe 9.3.2).

La génération de code `Python` s'effectue par une conversion de l'arbre syntaxique abstrait `Sophrosyne` vers un arbre syntaxique abstrait `Python`. De là, un programme (chaîne de caractères) est généré. Pour faciliter l'intégration applicative de la bibliothèque `sophrosyne`, une API REST permet d'interagir avec ces fonctions de traduction.

La bibliothèque `pysophrosyne` gère tous les aspects logiques liés à l'exécution des missions. La transformation de `Sophrosyne` vers `Python` en est d'autant simplifiés, toutes ces fonctionnalités étant fournies par l'héritage au sens de la programmation objet. Le taille du code résultant est linéaire en la taille du code `Sophrosyne`. À titre d'illustration, les programmes générés pour le système et la mission présentés au paragraphe 4.4 sont donnés ci-dessous.

```
class UA(System):
    position = StateVar({'x': 0, 'y': 0, 'z': 0})
    velocity = StateVar({'x': 0, 'y': 0, 'z': 0})
    acceleration = StateVar({'x': 0, 'y': 0, 'z': 0})
    time = StateVar(0)
    dtime = StateVar(1)
    bat = StateVar(100)

    def takeoff(self, alt):
        self._ = 0

    def orbit(self, x, y, z, r):
        self._ = 0

    def goto(self, x, y, z):
        self._ = 0
        self.start = self.system.position

    def land(self):
        self._ = 0
        self.startz = self.system.position['z']
```

Le programme généré pour un système est plus ou moins une coquille vide : les interfaces sont esquissées, mais il reste à implémenter les actions et la mise à jour de l'état du système.

```
class Context0(Context):

    def _guard(self):
        return self.system.bat > 30

    def main(self):
```

```
        yield from self.system.goto(0, -5, 10)
        yield from self.system.orbit(-8, -5, 10, 8)
        yield from self.system.goto(-3, 50, 22)
        yield from self.system.orbit(-11, 50, 22, 8)
        yield from self.system.goto(15, 75, 13)
        yield from self.system.orbit(7, 75, 13, 8)
        yield from self.system.goto(3, 50, 27)
        yield from self.system.goto(0, 0, 15)
        self.unregister()

    def fallback(self, cont):
        yield from self.system.goto(self.system.position['x'], self.system.
                                    position['y'], 30)
        yield from self.system.goto(5, 45, 30)


class _Main(Mission):

    def main(self):
        yield from self.system.takeoff(10)
        self.system.mission = Context0(self)
        yield next(self.system.mission)
        yield from self.system.land()
```

Le code de mission en revanche est complet : les fondements de son exécution seront expliqués lors de la présentation de `pysophrosyne` au paragraphe 6.2. On remarque que le code ainsi généré est « lisible » ; cette caractéristique est particulièrement importante pour le système, car il nécessite une intervention humaine pour le compléter. Pour conclure sur ces aspects de génération que nous ne détaillerons pas davantage, notons que la version symbolique du système ne nécessite elle aucune intervention : les invariants, prérequis et propriétés des actions sont convertis en des équations `Python` à l'aide de la bibliothèque `sympy` [186] ou en des modèles de logique dynamique différentielle.

6.2 Le modèle d'exécution de mission avec `pysophrosyne`

Un programme `Sophrosyne` a vocation à être utilisé par un système cyber-physique réel pour effectuer la mission envisagée. L'exécution doit implémenter les particularités de `Sophrosyne`, telles que la gestion des continuations. Nous proposons pour ce faire une bibliothèque `Python` servant à assurer cette exécution, `pysophrosyne`. Après avoir justifié la séparation du modèle analytique et de l'exécution, nous donnerons un aperçu de l'implémentation de la logique de `Sophrosyne` dans `pysophrosyne`. Celle-ci s'est progressivement complexifiée pour garantir le passage à l'échelle : une implémentation naïve risquerait par exemple de ne pas supporter la multiplication des contextes. Il est alors légitime de s'interroger sur les performances résultantes. Celles-ci étant difficilement estimables par la seule présentation de l'implémentation, nous montrerons une analyse du temps de réponse d'une exécution avec `pysophrosyne`.

6.2.1 Abandon du modèle analytique

Dans le chapitre 4, une action en `Sophrosyne` était présentée avec deux constituants : un prérequis, et un système différentiel. Pour expliciter la sémantique, nous avons déjà écarté les prérequis : ce ne sont que des auxiliaires pour la vérification de mission. Il nous reste à présent à nous séparer des systèmes différentiels, ce paragraphe devrait en donner la motivation.

La première raison est pragmatique : les systèmes différentiels sont le résultat d'un processus de modélisation et sont en cela susceptibles de ne pas traduire l'évolution réelle du système cyber-physique. Considérons à titre d'exemple la relation liant la vitesse à la position. La position d'un drone est calculée par un filtre de Kalman sur des données provenant du module de géolocalisation, de l'odométrie, de la barométrie, *etc.* La vitesse est obtenue à partir de la centrale inertielle, et de la géolocalisation. Il est dès lors peu probable que ces données de capteurs entretiennent un lien aussi précis que la relation différentielle du modèle analytique.

Les travaux cherchant à réutiliser leur modèle analytique sur le système réel effectuent alors de la vérification en ligne de la fidélité du modèle : ils découvrent ainsi le moment où l'exécution diverge [14] de l'évolution prévue par le modèle. S'il est appréciable pour le système cyber-physique de savoir qu'il ne sait plus, cette connaissance doit aller de pair avec un processus décisionnel ; que faire quand le modèle est invalidé par l'exécution ? Nous voyons quatre réponses possibles :

1. interrompre l'exécution
2. déclencher une procédure exogène
3. déclencher une procédure endogène
4. persévérer

Interrompre l'exécution en terminant le processus de manière inattendue ne paraît pas acceptable pour un système critique en premier lieu pour des raisons sécuritaires. Qu'advient-il du système cyber-physique ? La deuxième solution est de donner la main à un autre processus, à l'instar d'une procédure d'atterrisage d'urgence ou d'un télé-pilote. Nous repoussons cette idée pour maintenir une séparation des responsabilités : il est opportun d'avoir un module de supervision de l'exécution, qui est susceptible de faire basculer le système dans un autre mode de fonctionnement, mais nous préférons le voir introduit comme un module externe plutôt que d'en faire un mécanisme au sein de l'exécution de mission. En effet, d'autres vérifications peuvent s'avérer nécessaires (*e.g.* vérifier l'état de santé de l'acteur qui reprendra l'exécution à son compte), et celles-ci ne font pas partie des prérogatives de la mission. En revanche, il serait possible d'effectuer un branchement dans la mission, ce qui constitue la troisième proposition. `Sophrosyne` incorpore ce mécanisme au niveau de la mission, par ses structures de supervisions que sont les contextes. Cette supervision ne figure pas au niveau du système, les raisons d'avoir un modèle infidèle à l'exécution étant bien trop nombreuses. Une partie de ces raisons mènent d'ailleurs à des écarts bénins : ce sont par exemple des phénomènes transitoires, comme une bourrasque de vent ou une erreur d'un capteur. Dans ces cas, la réponse naturelle est de persévérer, de poursuivre l'exécution, et l'incohérence entre le modèle et l'exécution se réduit à une information superfétatoire puisque sans conséquence sur la décision.

6.2.2 Le langage d'implémentation : `Python`

La présentation précédemment faite de la sémantique semblait inviter à l'implémentation de `Sophrosyne` dans un langage de programmation disposant de continuation explicites, à l'instar de

14. Pour plus d'exactitude, il faut comprendre ici « diverge de manière observable » : la supervision en ligne permet de dire qu'il existe une évolution du modèle qui interpole la trace de l'exécution, mais elle ne permet pas d'affirmer la fiabilité du modèle

Ruby, Scheme, ou C (avec le couple d'instructions setjmp - longjmp). Toutefois, nous n'avons retenu aucun de ces langages ; notre choix s'est porté sur Python. À cela, plusieurs raisons : Python est généralement considéré comme *accessible* par rapport à d'autres langages de programmation. Les langages fonctionnels par exemple tirent profit d'une population rôdée aux mathématiques. Dans la mesure où il reste nécessaire d'écrire en partie l'interfaçage dans le langage destination, il nous apparaît opportun de choisir un langage présentant une faible barrière à l'entrée. Ce point n'est que renforcé par la deuxième raison : Python est aujourd'hui l'un des langages de programmation les plus utilisés, et comprend à ce titre une kyrielle de bibliothèques qui seront utiles à l'utilisateur pour l'interfaçage et pour le traitement intermédiaire des données des capteurs (*e.g.* pymavlink, rospy, ou même des bibliothèques de *machine learning* telles que scikit learn).

L'absence de continuations explicites est un problème qui se contourne aisément. L'équivalent le plus direct consisterait à examiner la pile d'exécution afin de récupérer l'état courant de l'exécution. Une autre solution, plus esthétique, utilise simplement des structures de contrôle de flux telles que des générateurs qui fournissent par définition des continuations réifiées. Nous en rappelons brièvement le fonctionnement ici, pour une présentation plus détaillée de la sémantique de Python, nous renvoyons par exemple vers [155, 19].

Une fonction générateur, caractérisée par l'emploi de l'instruction yield, renvoie un itérateur de générateur. Dans l'itérateur de générateur, chaque yield interrompt l'exécution, conservant l'état, duquel l'exécution reprendra au prochain appel d'itération. Dans l'example suivant, la fonction fibonacci permet ainsi de produire la séquence infinie des nombres de Fibonacci ; leur génération se fera par l'instruction à next(fibo).

```
def fibonacci():
    x, y = 0, 1
    while True:
        yield x
        x, y = y, x + y

fibo = fibonacci()
next(fibo)  # renvoie 0
next(fibo)  # renvoie 1
```

Un générateur peut également produire un itérateur fini, auquel cas les appels excédentaires lèvent une exception StopIteration. Si le générateur comprend une instruction return e, la première levée de StopIteration renverra l'expression e. Ce mécanisme nous sera utile pour passer des continuations.

```
def one_two():
    yield 1
    yield 2
    return "done"
it = one_two()
next(it)  # 1
next(it)  # 2
next(it)  # StopIteration: "done"
```

6.2.3 Le passage de continuation et la trampolinisation

Une pratique courante pour maîtriser la taille de la pile d'exécution lorsque l'on travaille avec des appels de fonctions imbriquées (en particulier dans la définition de fonctions récursives) est

la *Tail-Call Optimization*. Celle-ci permet d'éliminer les *frames* non nécessaires lors d'un appel terminal : la fonction appelante renvoyant simplement le résultat de la fonction appelée, il n'est pas nécessaire de multiplier les *frames*. Dès lors, il nous suffit de réécrire le programme sous forme de passage de continuation pour linéariser l'exécution.

```
def f(_continuation):
    # Corps de la fonction
    return _continuation()
```

`Python` ne fournit toutefois pas cette optimisation [15]. Il existe alors bien des moyens de l'émuler, la trampolinisation comptant parmi les plus esthétiques. Celle-ci enveloppe l'évaluation terminale au sein d'une fonction qu'elle renvoie, dont l'exécution est déléguée à un trampoline qui se charge d'exécuter tous les appels.

```
def trampoline(func):
    current_func = func
    while True:
        if callable(current_func):
            current_func = current_func()
        else:
            return current_func
def f():
    print("f")
    return lambda: print("g")

trampoline(f) # affiche f puis g
```

Nos missions étant de générateurs, il nous faut adapter cette trampolinisation, par exemple :

```
def run(mission):
    mission = mission()
    while True:
        try:
            yield next(mission)
        except StopIteration as e:
            if isinstance(e.value, (types.GeneratorType, mission.Mission)):
                mission = e.value
            elif callable(e):
                mission = e.value()
            else:
                return e
```

6.2.4 La réification des missions

Nous disposons à présent d'une technique pour exécuter des missions séquentielles qui s'instancient en itérateurs. Il nous faut encore prévoir un mécanisme pour gérer le contrôle de flux d'exécution. Pour cela, il est nécessaire d'agir sur la continuation d'une mission, potentiellement

15. l'argument pour ne pas intégrer cette fonctionnalité à `Python` est la cohérence des traces de *frames* : la *TCO* éliminant des *frames*, l'utilisation du *debugger* peut être déroutante.

en cours d'exécution. L'ajout de telles fonctionnalités invite à nous séparer des objets fonctionnels prédéfinis : nous définissons une classe abstraite `Mission` qui implémente le pattern itérateur (cf figure 6.3). L'attribut `mission` est une liste de générateurs correspondant à la mission considérée. Dans le cadre d'une mission simple, comprenant une composition séquentielle d'actions, la liste se réduit à l'instanciation de la méthode `main`. Pour la gestion du flux d'exécution des contextes, une classe `Context` vient décorer cette classe `Mission`. Elle y ajoute les fonctionnalités propres aux contextes. Pour la garde, on retrouve :

- une abstraction `guard` permettant son évaluation
- la liste de ses dépendances `dependencies`, auprès desquels le contexte s'inscrira

Pour les événements afférents aux contextes

- deux méthodes pour s'inscrire `register` et se désinscrire `unregister` auprès des dépendances (voir *Observer* pattern [102])
- deux méthodes pour se mettre sur liste noire (`sleep`) et s'en retirer (`wakeup`)
- l'évaluation de la garde et la gestion des inscriptions auprès des dépendances (`update`)
- la reprise dans le cadre des contextes à reprise ou à label (`resume`)

La mise sur liste noire est déclenchée par un contexte supérieur : lorsqu'une garde est invalidée la mission de repli est déclenchée et tous les contextes de la mission principale peuvent être mis en sommeil.

6.2.5 Analyse de performances

La bibliothèque `pysophrosyne` sert à l'exécution d'une mission `Sophrosyne` sur un système réel. Sa pertinence est étroitement liée au respect d'exigences non-fonctionnelles, au premier rang desquelles nous trouvons le temps de réponse. Nous n'avons conduit aucune étude détaillée sur ces aspects, cependant une présentation succincte de quelques mesures de profilage sont utiles pour en voir les linéaments. Les données que nous présentons ont été recueillies à l'aide de *timers* et de profilage de code [16] dans des simulations à logiciel dans la boucle (pour plus de détails à ce sujet, nous renvoyons vers le paragraphe 9.3.1). L'exécution s'effectue au sein d'un conteneur *docker*, sur un ordinateur portable suffisamment performant pour inviter à la précaution avant de les étendre aux systèmes embarqués traditionnels. Toutefois, les tendances sont elles indépendantes de ces plateformes (nous utilisons l'implémentation usuelle `CPython` qui est *single threaded*).

Nous considérons une mission consistant en un décollage du drone, suivi d'un déplacement supervisé vers un point. La supervision vérifie que la position du drone ne dépasse pas une valeur seuil selon l'une de ses composantes, sinon un atterrissage est déclenché. Le déplacement demandé entraîne systématiquement le déclenchement d'une telle mission de repli. Nous avons choisi une garde impliquant la position car celle-ci est mise à jour à chaque itération de la boucle d'exécution ; dès lors les contextes seront notifiés (pattern *Observer*) et toutes les gardes seront réévaluées. Les points d'attention que nous soulevons sont de deux ordres :

1. le temps de réponse général du contrôleur
2. le temps de réponse de la mise à jour de l'état du système
3. le temps de réponse de la gestion de l'exécution, c'est-à-dire de l'actualisation des contextes

Le premier est naturellement l'indicateur principal à retenir, car lui seul détermine l'utilisabilité de cette bibliothèque d'exécution. Les deux autres indicateurs nous permettent d'estimer le comportement du système selon sa dimension propre. En effet, le temps de réponse général du contrôleur comprend également des dépendances à l'égard de l'implémentation de chaque action.

16. Nous avons à cet effet utilisé l'outil standard `cProfile` offert par `Python`.

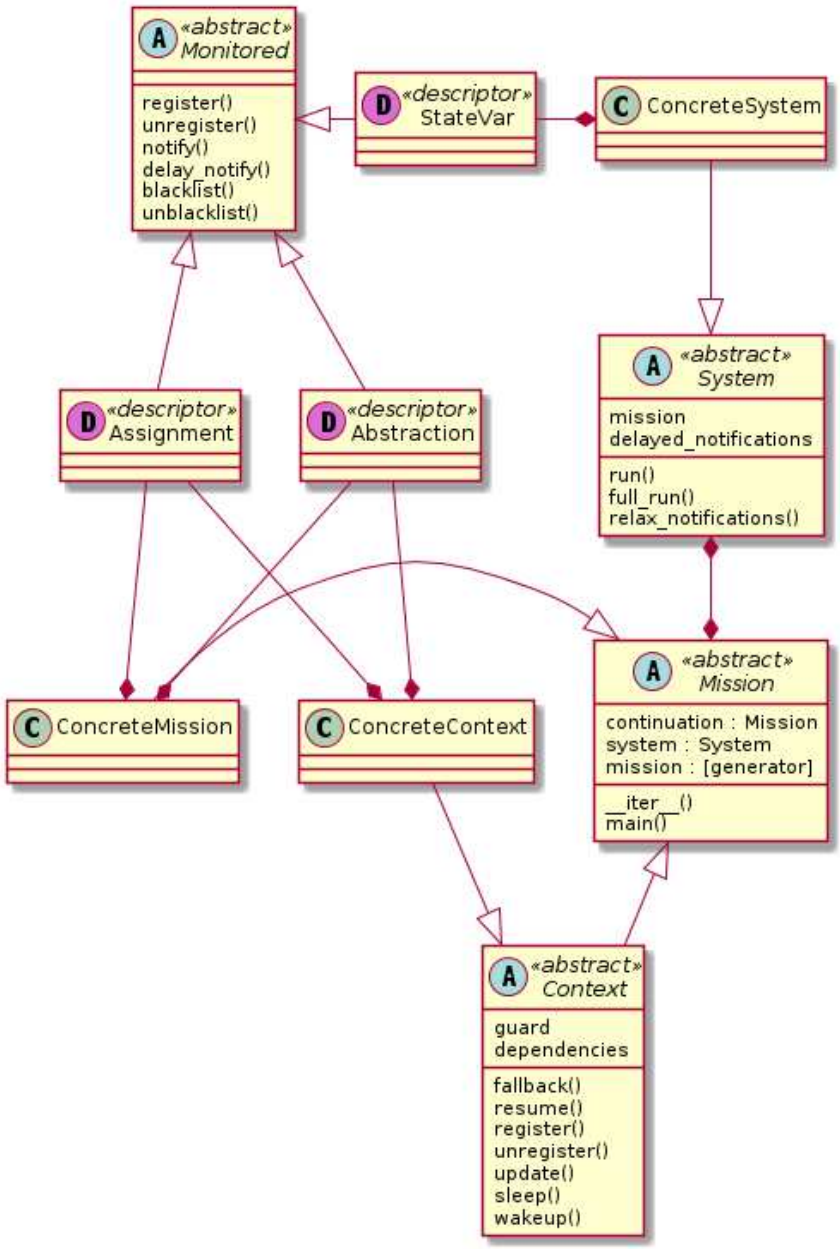


FIGURE 6.3 – Diagramme de classes de `pysophrosyne`

Toutes les actions que nous présentons sont sobres : elles comprennent tout au plus quelques évaluations de fonctions mathématiques usuelles (*e.g.* racine carrée) et d'expressions booléennes simples, ainsi qu'un envoi de message asynchrone. Nous en donnons les statistiques temporelles d'exécution dans la figure 6.4 : ces statistiques sont construites à partir des temps moyens d'exécution pour différentes simulations comprenant de zéro à cent contextes.

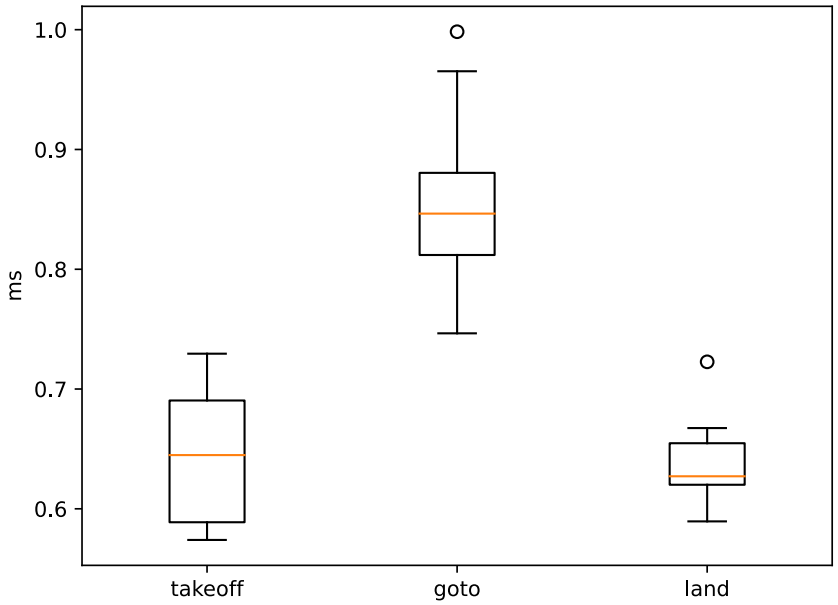


FIGURE 6.4 – Temps d'exécution moyen par appel des actions du système

Pour mesurer ces temps de réponse, trois fonctions sont observées :

1. la méthode `run` de la classe `System`, donnant le temps d'une itération (entre lesquels l'exécution est en pause pour 50 ms)
2. la méthode `update_state` de la classe `System`, donnant le temps de mise à jour des variables d'état (et leur écriture dans un journal d'exécution)
3. la méthode `relax_notifications` de la classe `System`, donnant le temps de mise à jour de l'état de tous les contextes

Intéressons nous dans un premier temps à l'influence du nombre de contextes sur le temps d'exécution. Les statistiques du temps de réponse d'une itération sont présentées dans la figure 6.5, et celles isolant le temps de réponse de l'actualisation des contextes sont reproduites dans la figure 6.6.

Conformément à ce que nous pouvions attendre, le temps de réponse croît linéairement avec le nombre de contextes actifs, et une régression linéaire est appropriée. Le poids que représente l'ajout d'un nouveau contexte est relativement faible, si bien que le temps de réponse général moyen ne double qu'à partir du centième contexte actif (cf. régression de la figure 6.5).

La variabilité des temps d'exécution nécessiterait une analyse appronfondie : l'utilisation de `Sophrosyne` pour une application temps réel peut s'en trouver compromise. Les pires temps d'exécution que nous constatons sur ces quelques tests pourraient s'avérer préjudiciables : on observe

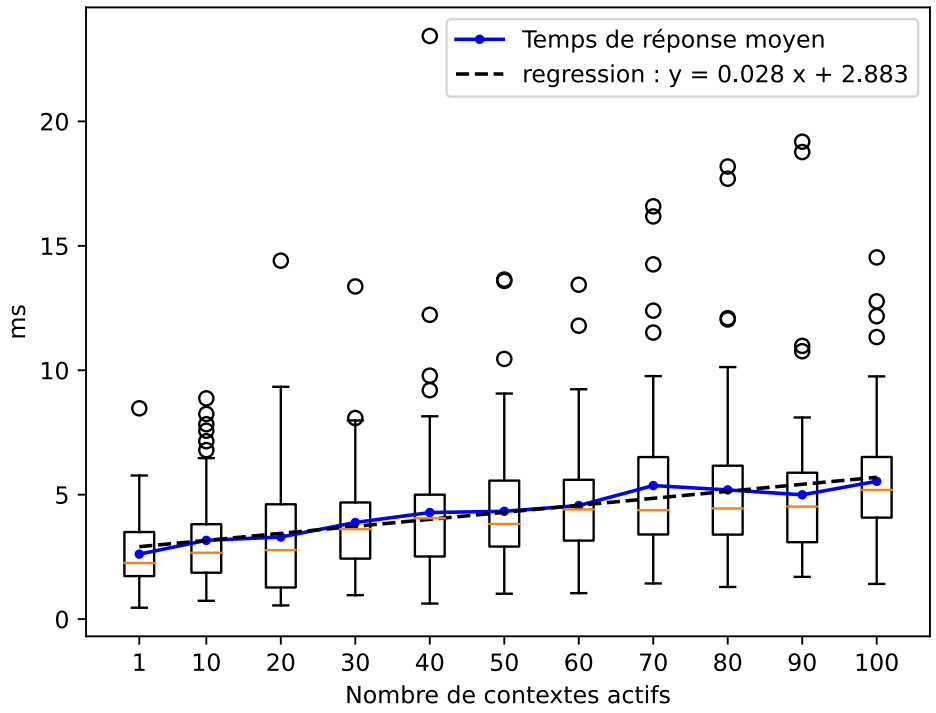


FIGURE 6.5 – Évolution du temps de réponse de la fonction `run` en fonction du nombre de contextes actifs.

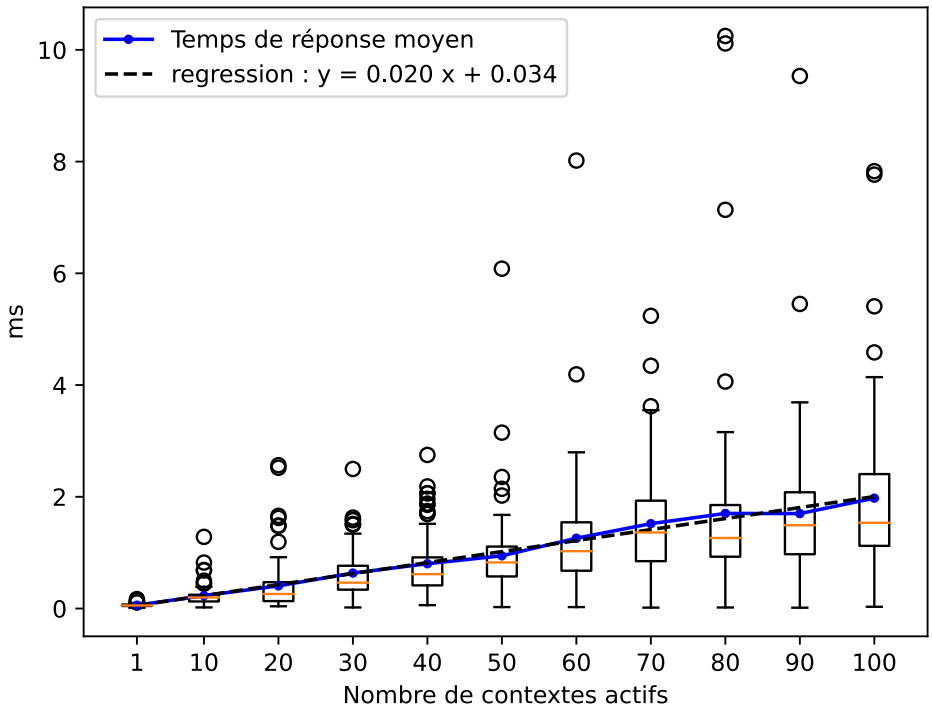


FIGURE 6.6 – Évolution du temps de réponse de la fonction `relax_notifications` en fonction du nombre de contextes actifs.

par exemple un pire temps environ six fois plus important que la médiane sur notre exécution avec quarante contextes (figure 6.5). Les sources de variabilités sont toutefois nombreuses : le cadre d'exécution ne se prête pas à une analyse du potentiel temps réel de la bibliothèque `pysophrosyne` (*e.g.* système d'exploitation non temps-réel, plateforme d'exécution virtualisée). De plus nous n'avons pas cherché à optimiser ces aspects, en désactivant le ramasse-miette de `Python` par exemple. En effet, ce temps de réponse n'a jamais été limitant pour les applications que nous avons développées, pour lesquelles une fréquence d'au plus 10 Hz était demandée [17].

Le délai introduit par la mise à jour des variables d'état est un potentiel facteur limitant. La figure 6.7 présente des données pour répondre à cela. Nous avons à cet effet mesuré le temps de réponse lors de l'exécution de la mission en introduisant de nouvelles variables. De même que la gestion des contextes dépend de la complexité de la garde à évaluer, le temps de mise à jour d'une variable d'état dépend nécessairement de la complexité de son calcul. Pour limiter cette influence sur la mesure, nous avons choisi de répliquer la variable de temps physique, de zéro à vingt fois. Le temps d'actualisation de ces variables est donc celui d'une lecture de la variable d'état temporelle et d'écriture de la variable d'état considérée. Cela correspond à la majorité des variables que nous avons rencontrées dans nos diverses expériences ; elles étaient fournies directement par l'autopilote.

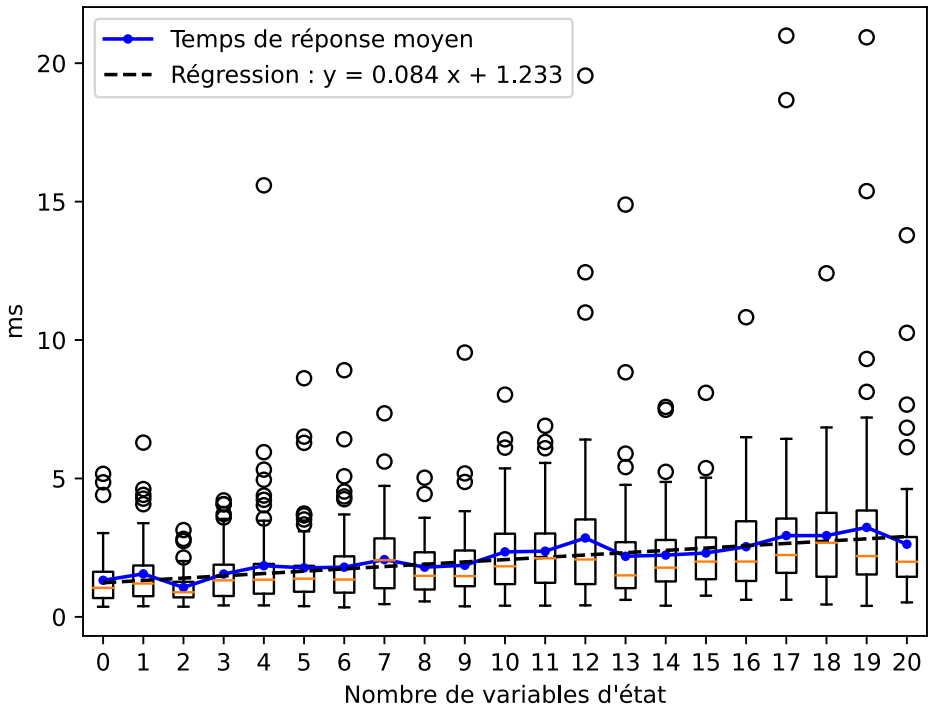


FIGURE 6.7 – Évolution du temps de réponse de la fonction `update_state` en fonction du nombre de variables d'état supplémentaires.

Ici encore, une évolution linéaire était attendue et les données s'y conforment. Le système

17. À titre de comparaison, le maintien du mode `offboard` par l'autopilote PX4, c'est-à-dire du contrôle par un ordinateur companion nécessite un flux de commande à au moins 2 Hz.

comporte de base (aucune variable d'état supplémentaire) huit variables d'état, dont quatre sont complexes et regroupent trois coordonnées (position locale, globale et du *home*, vitesse et attitude sous forme d'angles d'Euler). Cela fait donc un total de 18 valeurs numériques ou booléennes, pour lesquelles aucun calcul important n'est nécessaire bien que quelques traitements préalables soient effectuées avant leur écriture. L'ordonnée à l'origine de la régression semble ainsi compatible avec le taux d'accroissement. Le coût d'ajout de nouvelles variables d'état reste ainsi suffisamment faible pour envisager sereinement la majorité des applications (si tant est qu'aucun calcul lourd ne soit demandé). Sur les mesures présentées, nous observons une variabilité importante du temps de réponse à l'instar de celle que nous avons vu précédemment en augmentant le nombre de contextes actifs. Les pires temps d'exécution sont jusqu'à six fois supérieurs à ceux moyens. Cependant, comme nous l'avons indiqué précédemment, la plateforme de test n'était en aucun cas conçue pour des contraintes temps réel, et d'autres investigations doivent être entreprises si l'application envisagée est exigente.

6.3 Le modèle symbolique avec `symsophrosyne`

À ce point de notre développement, l'utilité des systèmes différentiels que nous introduisons lors de la modélisation d'un système reste encore à démontrer. En effet, la bibliothèque d'exécution ne les fait jamais intervenir, pas même comme moniteur de l'adéquation entre le modèle et l'exécution. Nous allons à présent mettre à profit ces descriptions, en introduisant la bibliothèque gérant ces aspects symboliques, `symsophrosyne`, et en développant les notions de correction de mission dans les chapitres qui suivront. L'action de cette bibliothèque intervient chronologiquement avant l'exécution de la mission. Elle n'est donc pas soumise aux mêmes impératifs de performance qu'un logiciel embarqué, et nous n'en détaillerons pas l'implémentation.

Le modèle analytique du système rend possible une réflexion sur la mission. Sous réserve que ce modèle capture bien la dynamique du système réel, il décrit pour le système l'ensemble des états accessibles par l'exécution de la mission. Cette information reste toutefois à compiler : initialement elle ne se présente que sous la forme d'une succession de systèmes différentiels. La première étape pour la valoriser consiste à expliciter l'état du système en tout point de la mission. Si nous sommes capables de mener à bien cette première tâche, les répercussions sont les suivantes :

- nous pouvons envisager des représentations graphiques de l'évolution de l'état du système, ce que nous présenterons avec les partitions de mission (cf paragraphe 6.3.2) et, dans une moindre mesure, avec l'interface graphique présentée au chapitre 9
- nous disposons de caractérisations précieuses de l'état du système qui serviront à la vérification formelle des missions (chapitre 8)

6.3.1 Le solveur sympy

Une des premières applications de l'informatique fut l'automatisation de calculs mathématiques (voir par exemple les calculateurs analogiques pour la résolution de systèmes d'équations différentielles ordinaires). Bien des progrès ont été faits entre ces premières réalisations et les usines à calculs que sont à présent les solutions logiciels telles que `MATLAB`, `Mathematica` ou `SageMath`. Pourtant, si ces développements sont fructueux sur le « dernier calcul », à l'instar d'une résolution d'équation identifiée comme linéaire, les heuristiques demeurent pour l'essentiel l'apanage du génie humain. L'ingénierie reste donc nécessaire afin de présenter les problèmes sous une forme qui convienne au système de calcul formel. Sans rentrer dans des détails inutiles, ce paragraphe illustre cette étape de mise en forme par différents traitements que nous appliquons.

Pour démontrer le raisonnement symbolique sur les missions `Sophrosyne`, nous avons choisi `sympy` [186]. Cette bibliothèque `Python` légère, comprise par ailleurs dans `SageMath`, présente l'avantage d'une intégration plus aisée du fait de l'adéquation des langages hôtes. La stratégie que nous avons mise en place pour préparer les actions à leur résolution par les solveurs `sympy` commence par la conversion des clauses en système d'équations lors de la génération du code du système symbolique depuis le programme `Sophrosyne`. Pour cela, nous introduisons les variables nécessaires ; l'inéquation `@z > 0` par exemple est transformée en `z(gamma) = nzpos0(gamma)` où `nzpos0` est une fonction de $[0, 1] \mapsto \mathbb{R}^{+*}$.

La deuxième étape se présente sous la forme d'une boucle pour laquelle chaque itération simplifie le système différentiel. Encore nous faut-il expliciter ce que nous entendons par « simple ». Notre objectif est d'obtenir une formulation de chaque variable d'état en fonction de γ, seulement cela n'est pas systématiquement possible. Observons par exemple un cercle, décrit par l'équation $x^2 + y^2 = r^2$. Nous ne serons généralement pas en mesure d'isoler les expressions d'une des variables $(x,\ y)$ de l'autre. Toutefois, les domaines peuvent être caractérisés : aussi, nous avons $x(\gamma) = -r + \text{pos0}\,(\gamma),\ \wedge\ x(\gamma) = r - \text{pos1}\,(\gamma)$ avec pos0, pos1 $\in [0, 1] \mapsto \mathbb{R}^{+}$. Bien d'autres calculs sont entrepris lors de cet étape, parmi lesquels nous retrouvons naturellement la résolution d'équations et d'équations différentielles, la déduction sur équations fonctionnelles, la résolution de systèmes, ou encore le calcul de limites. La figure 6.8 présente le système différentiel tel que produit par la première action de notre exemple, `takeoff(10)` (cf paragraphe 4.4) et le résultat obtenu à l'issue de la deuxième étape.

Lorsque cette étape est menée à bien, nous disposons d'une caractérisation directe des variables d'états par un système d'équation n'impliquant aucune équation différentielle ou variable d'état. Ces caractérisations seront précieuses pour la preuve de mission : nous verrons que les actions sont transformées en des boucles, dont ces caractérisations sont des invariants.

Nous ne développerons pas davantage l'implémentation de ce solveur de systèmes différentiels. À cela, deux raisons : *(i)* la fragilité de l'heuristique, *(ii)* l'absence de caractérisation. La fragilité de notre implémentation a pour conséquence des systèmes levant des exceptions, et des résolutions partielles. De plus, nous n'avons pas encore caractérisé ces cas, ce qui n'autorise pas la généralisation des portions pertinentes de notre algorithme. Nous avons néanmoins choisi d'évoquer cette implémentation quelque peu naïve d'un solveur afin d'illustrer une perspective concrète que nous voyons à notre travail.

Il nous paraît utile de reprendre cette démarche avec les considérations suivantes. Tout d'abord, la difficulté d'établir une heuristique pour automatiser des calculs est bien connue de la preuve formelle. À l'instar de ce qui est fait dans ce domaine (voir par exemple [78, 106, 101]), nous penchons pour le développement d'un langage de tactique pour la résolution de systèmes différentiels et plus largement pour le raisonnement symbolique et la production des partitions de mission `Sophrosyne`. Par pragmatisme, ce raisonnement doit notamment permettre une interaction avec l'utilisateur quand la tactique échoue. Enfin, nous avons précédemment signalé que la résolution d'une action par le solveur produit des invariants de boucles pour la preuve formelle : une intégration de ces invariants serait incontestablement un atout, cela s'apparente alors à utiliser ce solveur comme un *backend* de l'assistant de preuve (il intervient toutefois auparavant et non pendant la preuve).

6.3.2 Les partitions de mission

À partir des caractérisation obtenues par le solveur, nous pouvons calculer l'ensemble image de chaque variable d'état en substituant toutes les fonctions auxiliaires introduites en chemin. Cette étape nous amène également à faire des approximations numériques des nombres non

$$
\begin{aligned}
\operatorname{dtime}(\gamma) &= dt \wedge \\
\operatorname{vx}(\gamma) &= 0 \wedge \\
\operatorname{vy}(\gamma) &= 0 \wedge \\
\frac{d}{d\gamma} t(\gamma) &= \operatorname{dtime}(\gamma) \wedge \\
\frac{d}{d\gamma} \operatorname{vx}(\gamma) &= \operatorname{ax}(\gamma) \operatorname{dtime}(\gamma) \wedge \\
\frac{d}{d\gamma} \operatorname{vy}(\gamma) &= \operatorname{ay}(\gamma) \operatorname{dtime}(\gamma) \wedge \\
\frac{d}{d\gamma} \operatorname{vz}(\gamma) &= \operatorname{az}(\gamma) \operatorname{dtime}(\gamma) \wedge \\
\frac{d}{d\gamma} x(\gamma) &= \operatorname{dtime}(\gamma) \operatorname{vx}(\gamma) \wedge \\
\frac{d}{d\gamma} y(\gamma) &= \operatorname{dtime}(\gamma) \operatorname{vy}(\gamma) \wedge \\
\frac{d}{d\gamma} z(\gamma) &= 6 \times 10\gamma(1-\gamma) \wedge \\
\frac{d}{d\gamma} z(\gamma) &= \operatorname{dtime}(\gamma) \operatorname{vz}(\gamma) \wedge \\
\frac{d}{d\gamma} \operatorname{bat}(\gamma) &= -0.004 dt \times \\
&\quad (\operatorname{az}(\gamma) + 9.89 + (\operatorname{vx}(\gamma)^2 + \operatorname{vy}(\gamma)^2)^{0.5})
\end{aligned}
$$

$$
\begin{aligned}
\operatorname{ax}(\gamma) &= 0 \wedge \\
\operatorname{ay}(\gamma) &= 0 \wedge \\
\operatorname{az}(\gamma) &= -\frac{60(2\gamma-1)}{dt^2} \wedge \\
\operatorname{dtime}(\gamma) &= dt \wedge \\
t(\gamma) &= \gamma dt - \gamma_0 dt + t_0 \wedge \\
\operatorname{vx}(\gamma) &= 0 \wedge \\
\operatorname{vy}(\gamma) &= 0 \wedge \\
\operatorname{vz}(\gamma) &= -\frac{60(\gamma^2-\gamma)}{dt} \wedge \\
x(\gamma) &= x_0 \wedge \\
y(\gamma) &= y_0 \wedge \\
z(\gamma) &= -20\gamma^3 + 30\gamma^2 + 20\gamma_0^3 \\
&\quad - 30\gamma_0^2 + z_0 \wedge \\
\operatorname{bat}(\gamma) &= \operatorname{bat}_0 - 0.004 \times \\
&\quad \left(9.89 \times dt \times \gamma - \frac{60}{dt}\gamma(\gamma-1)\right)
\end{aligned}
$$

où $x(\gamma_0) = x_0 \ \wedge y(\gamma_0) = y_0 \ \wedge z(\gamma_0) = z_0 \ \wedge \operatorname{bat}(0) = \operatorname{bat}_0 \ \wedge dt \in [10, 12]$

FIGURE 6.8 – Système différentiel de l'action `takeoff(10)`. À gauche : système différentiel initial. À droite : après résolution par le solveur.

décimaux. En définitive, nous obtenons une représentation ensembliste de l'état du système selon l'avancement de la mission. Cette information brute est utile pour interroger des états précis, c'est-à-dire pour identifier toutes les composantes d'un état. Un exemple de question associée est : « Où peut se trouver le drone à cinq minutes dans la mission ? ». La réponse nécessite en effet de reformuler la description de l'état du système d'une paramétrisation en γ à une paramétrisation en le temps physique (et nécessite donc l'inversibilité de la caractérisation du temps).

Cette représentation conserve toutefois un caractère aride du fait de son expression analytique. Nous en proposons alors une version graphique plus facile d'accès : la partition de mission. Celle correspondant à notre exemple fil rouge (cf paragraphe 4.4) est donnée par la figure 6.9. Une partition de mission représente à la manière d'une partition d'orchestre l'évolution de chaque variable d'état en fonction de γ, où à chaque action correspond une mesure. Dans la palette de couleurs choisie, l'on retrouve :

- en vert une valeur exacte prise par la variable d'état
- en bleu un intervalle dans lequel la variable d'état prend sa valeur
- en orange les prérequis des actions entreprises

La partition représente ici l'évolution idéale. Nous lisons par exemple que la mission devrait durer entre 107 et 118 secondes. Concernant le contexte que nous avions défini pour surveiller le niveau de charge de la batterie, nous voyons que la seuil critique de 30% n'est jamais atteint. De même, les prérequis que nous avions ajoutés à chaque action sont validés : ceux-ci portent exclusivement sur la position. Lors d'une action `goto` ou `land`, le système doit être en vol, $z > 0$, et le drone doit être initialement en un point précis du cercle avant de commencer l'action `orbit`.

6.4 Conclusion

Après avoir présenté le langage dédié que nous avons développé pour la programmation de missions pour systèmes cyber-physiques dans les chapitres précédents, ce chapitre donne corps à ces concepts en présentant l'implémentation des principaux outils. Nous disposons de trois bibliothèques pour transformer, exécuter, et raisonner sur les missions `Sophrosyne`. La première, `sophrosyne`, gère tous les aspects liés à la compilation et les transformations entre modèles. La seconde, `pysophrosyne`, comprend le noyau logique nécessaire à l'exécution des missions. La troisième, `symsophrosyne`, initie le raisonnement sur le modèle analytique des missions, et en offre une analyse statique.

La bibliothèque `pysophrosyne` permet une écriture de programmes `Sophrosyne` en `Python` élégante : l'essentiel de la logique de gestion des continuations est dissimulée, sans pour autant que les performances n'en soient affectées de manière rédhibitoire. Nous avons montré que le temps de réponse d'une mission `Sophrosyne` ainsi implémentée était compatible avec des exigences de l'ordre de la centaine de hertz (sous réserve que les algorithmes intégrés soient eux-mêmes compatibles avec cette exigence), et le faible coût engendré par l'ajout de nouveaux contextes en fait une structure viable pour un large champs d'applications.

Les modèles analytiques définis pour chaque action permettent de simuler les états accessibles d'une mission, ce qu'implémente `symsophrosyne`. Cette bibliothèque en démontre l'utilisation et le potentiel au moyen des partitions de mission, et les produits intermédiaires de sont fonctionnement pourraient être réutilisés pour la preuve de mission que nous verrons aux chapitres 7 et 8. De nombreux efforts peuvent encore être envisagés pour généraliser son champ d'action : disposer d'un solveur universel sur tous les systèmes exprimables en `Sophrosyne` n'est qu'un idéal intouchable en l'état des connaissances.

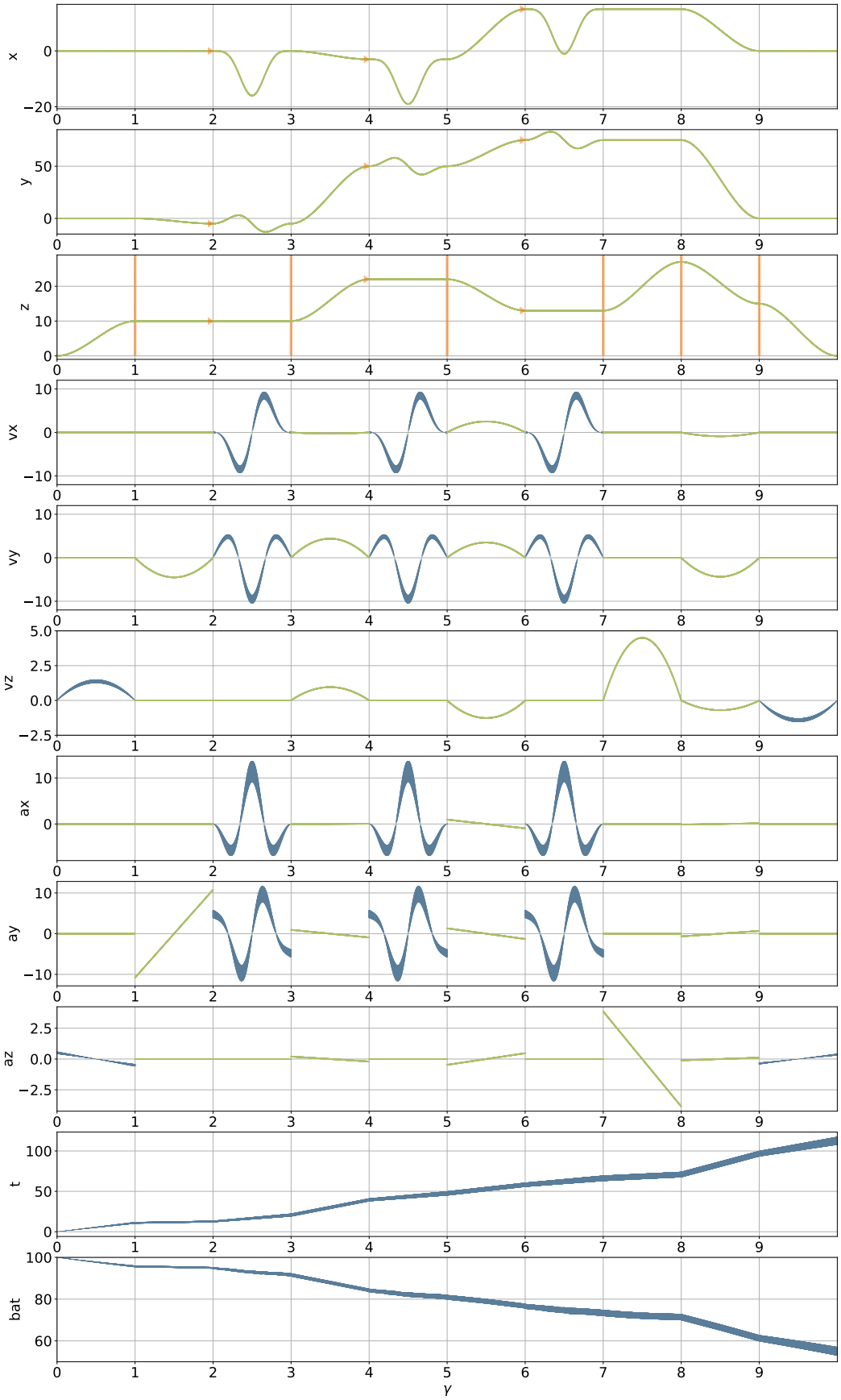


FIGURE 6.9 – Partition de la mission d'inspection de pylônes

Chapitre 7

De Sophrosyne à la logique dynamique différentielle

La vérification de programmes `Sophrosyne` nécessite un cadre formel. Celui que nous avons retenu est la logique dynamique différentielle $d\mathcal{L}$, principalement pour deux raisons. La première est la structure de la représentation : il nous est plus aisé de représenter les objectifs de preuves que nous dérivons des missions par des programmes hybrides que par de automates hybrides par exemple. La seconde raison est l'outillage autour de la logique dynamique différentielle : pour effectuer nos vérifications, nous utiliserons l'assistant de preuve `KeYmaera X` [10]. Ce chapitre présentera brièvement la logique dynamique différentielle avant d'exposer la transformation des concepts de `Sophrosyne` dans cette logique.

7.1 La logique dynamique différentielle

La logique dynamique différentielle ($d\mathcal{L}$) est une logique dynamique du premier ordre sur les réels. Elle est utilisée pour l'étude des programmes hybrides. Nous en rappelons ici la syntaxe au format Backus-Naur. Pour cela, nous considérons (éventuellement indicé) e un terme, x une variable, c une constante, χ une formule du premier ordre, ψ une formule, et α un programme hybride.

$$
\begin{array}{rr}
e ::= & x \mid x' \mid c \mid e_1 \mathcal{F} e_2 \mid (e)' \\
\alpha ::= & (x_i := e_i) \mid (x_i' = e_i) \& \chi) \mid ?\chi \mid \alpha_1 \cup \alpha_2 \mid \alpha_1 ; \alpha_2 \mid \alpha^* \\
\psi ::= & e_1 \mathcal{C} e_2 \mid \neg\psi \mid \psi_1 \mathcal{R} \psi_2 \mid \forall x \psi \mid \exists x \psi \mid \langle\alpha\rangle(\psi) \mid [\alpha](\psi)
\end{array}
$$

où $\mathcal{F} \in \{+, -, \cdot, /\}$, $\mathcal{C} \in \{=, \leq, \geq, <, >, \neq\}$ et $\mathcal{R} \in \{\wedge, \vee, \rightarrow, \leftrightarrow\}$.

L'essentiel de la sémantique est défini de manière habituelle, nous ne présentons par la suite que les éléments singuliers de la logique. Nous renvoyons vers [151] pour une présentation détaillée de la logique dynamique différentielle.

La $d\mathcal{L}$ dispose de deux opérateurs modaux pour la temporalité, $\langle\ \rangle$ et $[\ ]$:

- $\langle\alpha\rangle(\psi)$ signifie qu'il existe un état accessible par l'exécution du programme hybride α qui satisfait la formule ψ
- $[\alpha](\psi)$ signifie que tous les états accessibles par l'exécution du programme hybride α satisfont la formule ψ

L'opérateur $?\chi$ dénote un test conditionnel, son comportement est une non-opération si la formule χ est satisfaite ; dans le cas contraire, l'opérateur interrompt le programme. α; β correspond à la composition séquentielle de deux programmes. $\alpha \cup \beta$ est un choix non-déterministe : l'exécution suivra le programme α ou le programme β. α^* est une répétition non déterministe de programme : α sera exécuté zéro fois ou plus.

$(x' = e\&\chi)$ introduit une équation différentielle, où x' est la dérivée temporelle de x et χ une contrainte sur le domaine d'évolution. Une équation différentielle fait évoluer la variable dérivée. La *durée* d'évolution est non déterministe, toutefois elle ne peut pas invalider le domaine χ. Considérons un état ω. L'interprétation de la dérivation est donnée comme suit

$$[\![(e)']\!]_\omega = \sum_x \omega(x') \frac{\partial [\![e]\!]}{\partial x}(\omega)$$

avec les axiomes habituels

- $(e_1 + e_2)' = (e_1)' + (e_2)'$
- $(e_1 \cdot e_2)' = (e_1)' \cdot (e_2)'$
- $(c)' = 0$ pour une constante c
- $(x)' = x'$ pour une variable x

Enfin, la $d\mathcal{L}$ possède une affectation non-déterministe sur les réels, $x := *$. Cette opération est définie à partir des éléments précédents en utilisant le non-déterminisme de l'évolution différentielle : $x := 0; (x' = 1) \cup (x' = -1)$ Pour obtenir une valeur entre 0 et 1, nous écrirons ainsi : $x := *;\ ?0 \leq x \leq 1$

7.2 La modélisation du système cyber-physique

La logique introduite, nous pouvons à présent nous intéresser à la transformation de programmes `Sophrosyne` en modèles de $d\mathcal{L}$. À cet effet, cette section expose l'écriture de systèmes `Sophrosyne` en $d\mathcal{L}$.

7.2.1 Les systèmes

Un système de `Sophrosyne` est définit comme un tuple (Σ, $\mathcal{I}$, Ω) où

- Σ est l'ensemble des variables d'état
- $\mathcal{I}$ est l'ensemble des propriétés invariantes
- Ω est l'ensemble des actions exécutables par le système

Les variables d'état permettent de caractériser l'état du système, elles incluent par exemple l'altitude pour un aéronef. Le système est temporellement dépendant (ou dynamique) : à mesure qu'il exécute une mission, la valeur de ses variables d'état évolue. Leur domaine est soit celui des réels, soit celui des booléens. L'évolution d'une variable d'état est dictée par l'action alors exécutée (*e.g.* l'altitude croît quand l'aéronef décolle). Cette évolution doit toutefois satisfaire les propriétés invariantes, telles que les lois physiques (l'aéronef ne peut pas s'enfoncer dans le sol) ou que celles intrinsèques (lien entre la vitesse et l'accélération). Les variables d'état sont donc représentées par de simples variables en $d\mathcal{L}$.

7.2.2 Les actions

Une action est un programme hybride, qui définit l'évolution idéal du système. Elle mélange :

1. des transitions discrètes ω_d, considérées instantanées telles qu'un calcul de distance

2. une spécification différentielle ω_c décrivant l'évolution continue des variables dans un domaine ω_Δ

Les systèmes cyber-physiques que nous considérons exécutent un nombre fini d'actions, avec de potentielles répétitions. Cette restriction est non limitante quand le système cyber-physique consomme des ressources limitées. Le temps nécessaire à l'exécution d'une action demeure souvent approximatif, si ce n'est inconnu. Cette ambiguïté nous a amenés à introduire un argument principal γ, différent du temps physique, par rapport auquel les variables d'état réelles sont dérivables. Une relation persiste néanmoins entre γ et le temps physique t. Une dérivée du temps physique existe presque partout, et elle est à valeur strictement positive. La traduction en $d\mathcal{L}$ d'une action introduit systématiquement une période $\tau > 0$ dans l'exécution. En effet, plutôt qu'un modèle événementiel, les transitions sont déclenchées de manière temporelle. La période de ce déclenchement est non-déterministe, mais de pire cas τ. Cette période est à mettre en relation avec celle du contrôleur exécuté sur le système cyber-physique réel. Pour des raisons pratiques, nous dissocions la partie discrète de « mise en place » de l'action servant essentiellement à instancier des variables, du coeur de l'action ω. Ce dernier est le système hybride définissant l'évolution du système après initialisation de l'action jusqu'à complétion de l'action.

$$\omega \equiv \{t_0 := t;\, \omega_c \wedge \bigwedge_{\psi \in \mathcal{I}} \psi_c \wedge \gamma' = 1 \wedge \omega_\Delta \wedge \bigwedge_{\psi \in \mathcal{I}} \psi_\Delta \wedge t - t_0 < \tau \wedge \gamma \leq 1\}^* \tag{7.1}$$

Par la suite, les propriétés introduites par l'invariant ψ seront intégrées dans les propriétés des actions pour simplifier les notations. Dans la formule précédente (7.1), nous avons insisté sur l'existance de propriétés invariantes différentielles ψ_c et de domaine ψ_Δ. La variable γ est un indicateur de progression d'une action : il évolue entre 0, quand une action commence, et 1 quand elle se termine. L'exécution complète ininterrompue d'une action et assurée par les programmes de liaison introduits lors de la composition séquentielle, si bien qu'une action dans son environnement donne

$$\ldots;\ ?\gamma = 1; \gamma := 0;\ \omega_d;\ \omega;\ ?\gamma = 1;\ \gamma := 0;\ \ldots \tag{7.2}$$

L'évolution continue est décrite en utilisant des équations ou inéquations différentielles ω_c, et des fonctions implicites ω_Δ. Il est à noter que `Sophrosyne` et la $d\mathcal{L}$ ont deux approches différentes du problème du cadre : en phase d'évolution différentielle, $d\mathcal{L}$ considère les variables constantes sauf spécification contraire ; *a contrario*, `Sophrosyne` considère par défaut l'évolution des variables d'état comme non-déterministe, bien que dérivables par morceaux ($\exists d_x \in \mathbb{R}$ t.q. $x' = d_x$, ce que nous abrégeons par $x' = *$). Cet espace de fonctions est alors « creusé » pour obtenir les évolutions autorisées.

$$\begin{aligned} \text{Propriété d'action Sophrosyne} \mapsto \quad & \text{d}\mathcal{L} \\ x' > 4 \wedge y = 2x \wedge t' = 1 \mapsto \quad & \{y := 2x;\}\{t_0 := t; \\ & \quad x' > 4 \wedge t' = 1 \wedge y' = * \wedge \gamma' = 1 \\ & \quad \wedge y = 2x \wedge t - t_0 < \tau \wedge \gamma \leq 1\}^* \end{aligned}$$

Remarquons dans l'exemple précédent qu'une affectation est initialement effectuée au sein de ω_d pour les variables dont l'évolution est définie implicitement ($y = 2x$). En cas d'expression multi-variée, la convention est que l'affectation agit sur la première variable d'état apparaissant dans la clause, y dans l'exemple précédent. De plus, chaque clause est considérée séparément.

Les disjonctions permettent d'exprimer les transitions discrètes. Illustrons ce mécanisme avec une course : un véhicule cherche à franchir une ligne d'arrivée, sa destination. Cette action a de une à deux phases : le véhicule a une accélération constante `@A = 5`, et s'il atteint sa vitesse maximale `@Vmax = 20` l'accélération devient nulle. Sans perte de généralité, on suppose la position initiale (start) inférieure à la ligne d'arrivée (dest). Notons toutefois que la modélisation de plusieurs phases au sein d'une même action est un engagement fort : elle implique des transitions exogènes à l'instar du rebond déclenché par le sol pour une balle rebondissante. Dans le cas d'une transition assurée par le contrôleur que nous programmons, il convient de séparer chaque phase en une action dédiée.

```
invariant : {
  @t' > 0,
  @x' = @v * @t',
  @v' = @a * @t',
  @A = 5,
  @Vmax = 20
}

action course(dest):
  requires @x < dest
  start <- @x
  @x = (1 - γ) * start + γ * dest and
  (@v <= @Vmax and @a = A) or
  (@v >= @Vmax and @a = 0)
```

Le programme hybride $d\mathcal{L}$ correspondant est donné par le modèle 1. Notons que nous n'y avons pas inclus l'initialisation de l'action, qui comprend notamment la définition des variables `start` et `dest`.

`Sophrosyne` introduit également une structure de supervision d'une action. Dans ce cas, l'exécution de l'action est sujette au respect d'une propriété. Nous écrivons pour une action exécutée dans le contexte d'une garde ϕ :

$$\omega \odot \phi \equiv \{?\phi;\, t_0 := t;\, \omega_c \wedge \gamma' = 1 \wedge \omega_\Delta \wedge t - t_0 < \tau \wedge \gamma \leq 1\}^*$$

Conformément à l'intuition, une structure de supervision dont la garde est une tautologie se réécrit

$$\omega \odot \top \equiv \omega$$

Ainsi, si la garde n'est jamais invalidée, l'action s'exécute indifféremment en présence de cette structure de supervision. Si toutefois la garde est invalidée avant la complétion de l'action, l'exécution *sort* de l'action. Pour la présentation de la transformation d'une action `Sophrosyne` en un modèle $d\mathcal{L}$, nous nous contenterons pour l'instant de cette définition. Nous compléterons ces premiers éléments lorsque nous nous intéresserons aux structures de supervision (section 7.3.2).

Un dernier concept de l'action reste à aborder, celui des prérequis. La faisabilité d'une action dépend généralement de l'état du système : pour décoller, le drone doit être préalablement au sol. Une action ω est toujours définie avec des prérequis $\underline{\omega}$. $\underline{\omega}$ est une formule propositionnelle que l'état du système *devrait* satisfaire juste avant de débuter l'action (limite à gauche). Ces

Modèle 1: Modèle $d\mathcal{L}$ de l'action `course`

```
1  {
2  |  t_0 := t; {
3  |  |  ?v ≤ Vmax;   a := A;                                  #Accélération
4  |  |  {
5  |  |  |  ∃dt > 0(                                          #évolution du temps
6  |  |  |  γ' = 1, x' = v × dt, v' = a × dt, a' = 0, t' = dt)   #Description
   |  |  |                                                     #↪ différentielle
7  |  |  |  ∧γ ≤ 1 ∧t − t_0 < τ ∧x = (1 − γ) × start + γ × dest   #Description
8  |  |  |  ∧v ≤ Vmax                                         #↪ implicite
9  |  |  }
10 |  |  ∪{
11 |  |  |  ?v ≥ Vmax;   a := 0;                              #Vitesse constante
12 |  |  |  {
13 |  |  |  |  ∃dt > 0(                                       #Évolution du temps
14 |  |  |  |  γ' = 1, x' = v × dt, v' = a × dt, a' = 0, t' = dt)   #Description
   |  |  |  |                                                  #↪ différentielle
15 |  |  |  |  ∧γ ≤ 1 ∧t − t_0 < τ ∧x = (1 − γ) × start + γ × dest   #Description
16 |  |  |  |  ∧v ≥ Vmax                                      #↪ implicite
17 |  |  |  }
18 |  |  }
19 |  }
20 }*
```

prérequis sont des artéfacts des systèmes symboliques, mais ils ne contraignent pas l'exécution réelle. Nous reviendrons sur ces éléments quand nous aborderons la vérification de mission au chapitre suivant (section 8.1).

7.3 Les missions

Nous définissons l'ensemble des missions de `Sophrosyne` $\mathcal{M} \ni \alpha, \beta$ pour un système υ de manière inductive :

- *actions* : une action du système est une mission
  $\forall\{\omega_d; \omega\} \in \Omega, \{\omega_d; \omega\} \in \mathcal{M}$
- *computations* : un calcul pure est une mission
  $\forall\kappa \in \mathcal{K}, \kappa \in \mathcal{M}$
- *séquences* : une composition séquentielle de missions est une mission
  $\forall\alpha, \beta \in \mathcal{M}, \{\alpha; \gamma := 0; \beta\} \in \mathcal{M}$
- *contextes* : les missions sont composables par formules propositionnelles
  $\forall\phi \in \Phi, \phi^{\{\gamma:=0;\beta\}}(\alpha) \in \mathcal{M}$

Conformément à l'usage, nous appellerons *fonction* une mission n'impliquant aucune action. Les actions sont les uniques structures atomiques continues, ayant une épaisseur temporelle, toute autre opération est considérée instantanée. Par ailleurs, l'initialisation d'une action ω_d est traitée par la suite comme une computation.

7.3.1 La préparation des continuations et des contextes

Nous avons vu, lors de la présentation du langage (chapitre 4), trois syntagmes particuliers permettant d'introduire des non-linéarités dans l'exécution : `label L`, `resume L`, et `resume`. Nous avons remarqué que le concept sous-jacent à ces structures était la continuation, seulement la logique dynamique différentielle ne présente pas de continuation explicite. Pour les traiter, nous allons les réécrire, en les intégrant aux contextes auxquels ils se rapportent. Pour se faire et sans perte de généralité, considérons l'ensemble des contextes $\mathcal{C}$ comme ordonné, nous noterons $\phi_k^{\beta_k}(\alpha),\ k \in \mathcal{C} \subset \mathbb{N}$. Nous introduisons trois ensembles formant une partition de $\mathcal{C}$, $\mathcal{A} \cup \mathcal{L} \cup \mathcal{R} = \mathcal{C}$. $\mathcal{A}$ correspond à l'ensemble des contextes abortifs, $\mathcal{L}$ aux contextes à label, et $\mathcal{R}$ aux contextes à reprise.

7.3.2 La sémantique des contextes

Pour décrire la sémantique des contextes, nous opérons préalablement quelques réécritures. La première étape introduit un booléen, δ_k indiquant si un contexte abortif peut être exécuté.

$$\phi_k^{\beta_k}(\alpha) \mapsto \{\delta_k := \top\}\phi_k^{\beta_k}(\alpha) \qquad \text{si } k \in \mathcal{A}$$

Cette porte est initialement ouverte : la garde n'a pas été invalidée, et la mission de repli n'a pas encore été exécutée. Seule l'exécution de la mission de repli β_k fermera la porte δ_k. Un contexte abortif ($k \in \mathcal{A}$) portant sur une action se traduira alors en d$\mathcal{L}$ ainsi :

$$\begin{aligned}\phi_k^{\beta_k}(\omega_d) \equiv &\{?\delta_k;\ \{?\phi_k; \omega_d;\} \cup \{?\neg\phi_k;\}\} \cup \{?\neg\delta_k;\}\\ \phi_k^{\beta_k}(\omega) \equiv &\{?\delta_k;\ \{\omega \odot \phi_k\} \cup \{?\neg\phi_k;\ \beta_k;\ \delta_k := \bot\}\}^*\} \cup \{?\neg\delta_k\}\end{aligned}$$

L'introduction des portes nous permet donc de distribuer les contextes sur la mission principale :

$$\phi_k^{\beta_k}(\alpha_0;\ \alpha_1) \mapsto \phi_k^{\beta_k}(\alpha_0);\ \phi_k^{\beta_k}(\alpha_1)$$

de sorte que nous pourrons désormais considérer les contextes comme portant uniquement sur une action, une computation, ou un autre contexte.

La réécriture suivante permet de rapprocher les contextes à label des contextes à reprise. Avant de la présenter, il nous faut avertir que les missions de repli β_k contiennent les contextes dans lesquels elles s'exécutent ($\beta_i, \quad i < k$). Concrètement :

```
with φ0:
  with φ1:
    α
  else:
    α1;
else:
  α0;
```

$$\phi_0^{\beta_0} \circ \phi_1^{\beta_1}(\alpha) \quad \equiv \quad \phi_0^{\alpha_0} \circ \phi_1^{\phi_0^{\alpha_0}(\alpha_1)}(\alpha)$$

Pour tout label nous effectuons la réécriture suivante

$$\text{label L};\ \alpha;\ \phi_0^{\beta_0} \circ \ldots \circ \phi_n^{\beta_n}(\omega) \quad \mapsto \quad \alpha;\ \phi_0^{\beta'_0} \circ \ldots \circ \phi_n^{\beta'_n}(\omega)$$

où $\beta'_k := \begin{cases} \beta_k;\ ?\gamma = 1;\ \gamma := 0;\ \alpha; & \text{si } k \in \mathcal{L} \\ \beta_k \text{ sinon} \end{cases}$

Suite à notre remarque précédente, nous insistons sur le fait que les contextes qui englobaient β_k n'affectent pas α. En somme, cette transformation consiste en une itération de la boucle qu'introduit le contexte à label : tandis qu'avant α était vu comme un préfixe au contexte $\phi_k^{\beta_k}$, α est à présent un suffixe de la mission de repli β'_k. De là, la sémantique générale des contextes est donnée comme suit

$$\phi_0^{\beta_0} \circ \ldots \circ \phi_n^{\beta_n}(\omega_d) \equiv \{?\bigwedge_{k=0..n} \phi_k \wedge \bigwedge_{k\in\mathcal{A}} \delta_k;\ \omega_d;\} \cup \{?\neg(\bigwedge_{k=0..n} \phi_k \wedge \bigwedge_{k\in\mathcal{A}} \delta_k);\} \tag{7.3}$$

$$\phi_0^{\beta_0} \circ \ldots \circ \phi_n^{\beta_n}(\omega) \equiv \Big\{\{?\bigwedge_{k\in\mathcal{A}} \delta_k\};\ \{\omega \odot \bigwedge_{k=0..n} \phi_k\} \tag{7.4}$$

$$\cup \bigcup_{k\in\mathcal{A}} \{?\neg\phi_k \wedge \bigwedge_{i=0..k-1} \phi_i;\ \beta_k;\ \delta_k := \bot\} \tag{7.5}$$

$$\cup \bigcup_{k\in\mathcal{R}} \{?\neg\phi_k \wedge \bigwedge_{i=0..k-1} \phi_i;\ \gamma_k := \gamma;\ \beta_k;\ ?\gamma = 1;\ \gamma := \gamma_k\} \tag{7.6}$$

$$\cup \bigcup_{k\in\mathcal{L}} \{?\neg\phi_k \wedge \bigwedge_{i=0..k-1} \phi_i;\ \beta_k;\ ?\gamma = 1;\ \gamma := 0\} \tag{7.7}$$

$$\Big\}^* \tag{7.8}$$

Dans le modèle précédent, on retrouve une forme généralisée de l'exécution des missions de repli des contextes abortifs (7.5). Les clauses 7.6 et 7.7 correspondent respectivement aux missions de repli des contextes à reprise et des contextes à label. Suite à la réécriture précédente, la différence se trouve dans la valeur de γ restaurée : dans un contexte à reprise, il s'agit de reprendre dans l'état d'avancement laissé tandis qu'un contexte à label reprend l'action du début.

7.4 Conclusion

Dans ce chapitre, nous avons commencé par nous familiariser avec la logique dynamique différentielle : celle-ci permet de modéliser des programmes hybrides et de les analyser. Nous avons ensuite présenté les transformations permettant de passer de programmes `Sophrosyne` à des programmes hybrides de la logique dynamique différentielle. Le volet analyse a toutefois été occulté. La vérification de modèles en logique dynamique différentielle repose sur un calcul de séquents dont les règles sont présentées dans [151] et reproduites dans l'annexe B. Nous justifions notre choix de ne pas les développer ici par l'interaction que nous avons avec cette logique. L'utilisation que nous en montrerons dans cet ouvrage (cf chapitre 8) s'arrête pour l'essentiel à la modélisation. Nous nous limiterons en effet à l'exposition des objectifs de preuve pour vérifier la correction d'une mission `Sophrosyne`, et si une preuve concrète est évoquée en guise d'illustration, l'absence d'un développement méthodique sur les axiomes de la logique dynamique différentielle ne devrait pas en empêcher la compréhension.

Chapitre 8

Sophrosyne : la vérification de mission

Et, par principe, nous inclinons à prétendre que les jugements les plus faux (dont les jugements synthétiques a priori font partie) sont, pour nous, les plus indispensables [...]

Par delà le bien et le mal, Nietzche

Les développements précédents ont donné à `Sophrosyne` une teneur logique que nous allons mettre à profit. Les questions que nous nous posons dorénavant concernent la correction d'une mission :

- la mission est-elle exécutable ?
- le comportement est-il celui attendu ?

Rappelons encore une fois qu'en toute rigueur ces deux questions restent généralement dépourvues de réponse certaine : nous serons tout au plus à même de détecter des erreurs patentes, si nous oublions de faire décoller le drone par exemple. Cette conviction est au coeur de notre méthode, et c'est pourquoi l'on ne s'étonnera pas de chercher à prouver que des missions de repli ne serons jamais déclenchées ; ces preuves valent pour le modèle mais la réalité est souvent bien plus capricieuse.

Conformément à l'usage, nous parlerons de propriété de vivacité pour le caractère exécutable d'une mission, et de sûreté pour l'adéquation du comportement. Ce dernier terme prendra tout son sens quand nous détaillerons les objectifs de preuve.

8.1 La vivacité

Une mission est une succession d'actions qui confèrent au système des dynamiques variées. La transition d'une action à une autre survient dans deux cas, *(i)* lorsque l'action est complétée, ou *(ii)* lorsque qu'une garde devient invalide. La propriété de vivacité, indispensable, s'intéresse à la faisabilité d'une mission. Deux notions complémentaires y concourent :

- la *vivacité atomique*, qui vérifie que le système n'atteint jamais un état bloqué, c'est-à-dire sans transition
- la *vivacité séquentielle*, qui interroge la satisfaction des prérequis de chaque action

8.1.1 La vivacité atomique

La vivacité atomique est une propriété locale : elle s'évalue au niveau d'une action, en vérifiant qu'une transition vers une autre action se produira.

Vivacité atomique. *La vivacité atomique* Λ *est définie par induction*

$$\Lambda(\omega;\,\alpha) \equiv [\omega]\langle\omega\rangle\big(\gamma = 1 \wedge \Lambda(\alpha)\big) \tag{8.1}$$

$$\Lambda(\kappa;\,\alpha) \equiv [\kappa]\Lambda(\alpha) \tag{8.2}$$

$$\Lambda(.) \equiv \top \tag{8.3}$$

$$\Lambda(\phi_0^{\beta_0} \circ \ldots \circ \phi_n^{\beta_n}(\omega);\,\alpha) \equiv \Big[\phi_0^{\beta_0} \circ \ldots \circ \phi_n^{\beta_n}(\omega)\Big]\Big\langle\phi_0^{\beta_0} \circ \ldots \circ \phi_n^{\beta_n}(\omega)\Big\rangle \tag{8.4}$$

$$\Big((\gamma = 1 \wedge \Lambda(\alpha)) \vee \tag{8.5}$$

$$\left(\bigvee_{k=0..n} \neg\phi_k \wedge \bigwedge_{i<k} \phi_i \wedge \Lambda(\beta_k)\right)\Big) \tag{8.6}$$

$$\Lambda(\phi_0^{\beta_0} \circ \ldots \circ \phi_n^{\beta_n}(\kappa);\,\alpha) \equiv \bigwedge_{k=0..n} \phi_k \wedge [\kappa]\,\Lambda(\alpha) \vee \bigvee_{k=0..n} \neg\phi_k \wedge \bigwedge_{i<k} \phi_i \wedge \Lambda(\beta_k) \tag{8.7}$$

Remarque : *nous supposons ici, afin de faciliter la lecture, que l'indexage des contextes composés décrit les entiers de* 0 *à* n*. En toute rigueur il n'en est qu'un sous-ensemble, et l'on retirera de part et d'autre les contextes inactifs.*

Cette définition nous donne des réécritures équivalentes de la propriété de vivacité atomique. La réécriture 8.3 correspond au cas d'arrêt. Une computation n'impliquant pas d'action, elle met à jour la configuration pour interroger la vivacité atomique de la suite de la mission (réécriture 8.2). L'action constitue le coeur de la vivacité : dans son cas simple (réécriture 8.1), nous vérifions que l'action se termine, ce qui se réécrit de manière équivalente par « pour toute exécution partielle de l'action, il existera une exécution la menant à son terme ». Le paramètre γ nous fournit un moyen simple d'évaluer la complétion de l'action. Pour une action complexe, c'est-à-dire environnée de contextes, nous distinguons deux cas :

- (8.5) l'action se termine normalement, il n'y a ici aucune différence par rapport au cas précédent de l'action simple
- (8.6) durant l'exécution de l'action, une garde devient invalide, auquel cas nous vérifions la vivacité atomique de la mission de repli

Enfin, (8.7) traite du cas d'une computation environnée de contextes. Le principe est le même que pour une action complexe.

Avant de mettre en application cette définition, observons dans quels cas la propriété de vivacité atomique n'est pas vérifiée. Pour une action isolée, en-dehors de tout contexte, l'absence de terminaison peut provenir de deux situations.

- le système différentiel diverge : ce serait le cas si notre action comporte par exemple

$$\text{time}' = \frac{1}{1-\gamma}$$

donnant au temps une allure logarithmique ($\text{time} = \text{time}_0 - \log(1-\gamma)$), définie sur $[0,\ 1[$. L'exemple que nous donnons ici implique clairement un problème de définition de l'action, mais dans des cas plus subtils, l'émergence de ces comportements dépend des conditions initiales.

- la frontière du domaine est atteinte : la consommation d'une ressource finie ou la rencontre avec un obstacle en sont archétypiques, par exemple

$$\text{battery} := 0.5; \{\text{battery}' = -1 \wedge \text{battery} \geq 0\}$$

  Pour l'essentiel, ces cas traduiront des incohérences physiques, dont l'expression transparaîtra dans le domaine d'évolution.

Le programmeur peut alors s'affranchir de cette terminaison de mission en englobant l'action dans un contexte. En effet, nous avons vu qu'invalider une garde était une condition suffisante pour la vivacité atomique, de sorte que l'on modifiera généralement l'exemple précédent ainsi :

$$\text{battery} := 0.5; \{\text{battery}' = -1 \wedge \text{battery} \geq 0 \odot \text{battery} > 0.2\}$$

Cette utilisation des contextes est particulièrement critique pour assurer la sortie des boucles que sont les contextes à reprise.

8.1.2 La vivacité séquentielle

Le second composant de la vivacité est la vivacité séquentielle. Celle-ci porte sur l'enchaînement des actions, en vérifiant que toutes les conditions sont réunies pour amorcer une nouvelle action. Ces conditions correspondent aux prérequis que nous avons rencontrés lors de la présentation générale de `Sophrosyne` (cf. Chapitre 4). Nous nous demandons donc si l'exécution des préfixes d'une mission satisfait les prérequis de l'action qui suit. Cette propriété concerne évidemment la composition séquentielle, mais également la composition par contexte. Trois transitions existent autour d'un contexte :

1. avec la mission suffixe, lorsque l'exécution a préservé la garde
2. avec la mission de repli, lorsque l'exécution a invalidé la garde
3. avec le point de retour à l'issue de l'exécution de la mission de repli, que cela soit la mission suffixe, un label, ou l'action qui était en cours d'exécution.

Les prérequis d'une action décrivent l'état attendu du système avant l'exécution d'une action : ils ne concernent donc pas le point de retour des contextes à reprise. En effet, illustrons ce point par une action de décollage, paramétrée par l'altitude à atteindre. Le prérequis que nous considérons est que l'aéronef doit être au sol avant de débuter l'action. Supposons qu'au milieu du décollage, alors que l'aéronef est à mi-hauteur, une bourrasque le déporte. En prévision de cet événement, nous avions défini un contexte circonscrivant dans le plan la position acceptable du système cyber-physique. La mission de repli, que nous ne détaillons pas, ramène l'aéronef à la verticale de son point de décollage et l'action peut reprendre. Clairement, nous ne nous attendons pas à ce que l'aéronef soit au sol pour reprendre cette action.

Avant d'introduire la vivacité séquentielle, il nous faut encore étendre la notion de prérequis d'une action aux prérequis d'une mission. Les prérequis portent toujours sur le préfixe de la mission : par induction il nous suffit donc d'expliciter ces prérequis selon que l'entête est une action, une computation, ou un contexte. Ces différents cas sont présentés dans le tableau 8.1. Arrêtons-nous un instant sur les prérequis d'un contexte : nous y retrouvons le branchement qu'introduit cette structure. La bonne transition vers un contexte dépendra de la validité de la garde : si elle est satisfaite, les prérequis de la mission principale doivent l'être également, sinon il faut vérifier ceux de la mission de repli.

Cette définition des prérequis établie, nous pouvons à présent introduire la vivacité séquentielle.

TABLE 8.1 – Définition des prérequis d'une mission

mission	prérequis
$\omega;\ \alpha$	$\underline{\omega}$
$\kappa;\ \alpha$	$\top$
$\phi^{\beta}(\alpha);\ \alpha_1$	$(\phi \wedge \underline{\alpha}) \vee (\neg\phi \wedge \underline{\beta})$

Vivacité séquentielle. *La vivacité séquentielle Z d'une mission est définie par induction*

$$Z(\kappa;\ \alpha) \equiv [\kappa]\big(\underline{\alpha} \wedge Z(\alpha)\big) \tag{8.8}$$

$$Z(.) \equiv \top \tag{8.9}$$

$$Z(\omega;\ \alpha) \equiv [\omega] Z(\alpha) \tag{8.10}$$

$$Z(\phi_0^{\beta_0} \circ \ldots \circ \phi_n^{\beta_n}(\omega);\ \alpha) \equiv [\phi_0^{\beta_0} \circ \ldots \circ \phi_n^{\beta_n}(\omega)] Z(\alpha) \tag{8.11}$$

En définitive, la vérification de la vivacité séquentielle non dégénérée survient toujours après l'exécution d'une computation. En effet, elle porte sur le coeur d'une action et s'effectue donc après l'exécution de l'initialisation de celle-ci.

8.1.3 La vivacité d'une mission

La vivacité atomique nous informe sur la complétion des actions, celle séquentielle sur les transitions au sein des missions. Les deux sont nécessaires pour assurer la correction d'une mission. Il nous reste à présent à formuler la vivacité générale d'une mission.

Vivacité d'une mission. *Une mission α est dite vivace quand elle satisfie les propriétés de vivacité atomique et de vivacité séquentielle, i.e. pour un état initial ψ_{init}, nous voulons*

$$\psi_{init} \vdash \Lambda(\alpha) \wedge Z(\alpha)$$

8.1.4 Une illustration de la vivacité

Mettons cette définition en pratique sur le début d'une mission drone. Nous allons vérifier que le décollage se déroule correctement du point de vue de la vivacité atomique. Pour cela, considérons comme premier modèle de notre système le drone décrit précédemment (voir chapitre 4), sans l'indicateur de charge de la batterie. Nous nous intéressons à un décollage à 10 mètres, et le modèle correspondant est le suivant. Dans celui-ci, nous avons renommé les variables composées (par exemple `position.z` devient p_z) et effectué les réécritures nécessaires pour que la syntaxe soit compatible avec `KeYmaera X`. En particulier, nous sommes contraint d'introduire des résultats du solveur de `symsophrosyne` pour contourner certaines limitations de `KeYmaera X`, à l'instar de l'absence d'évolution non-déterministe ou de fonction puissance rationnelle.

Théorème : vivacité atomique du décollage. *L'action `takeoff` de la mission d'inspection satisfait la propriété de vivacité atomique : le séquent présenté dans le modèle 2 est satisfait.*

Démonstration. Le détail de la preuve étant fastidieux, nous en donnons ici les grandes lignes. L'opérateur boîte est remplacé par un invariant de boucle : $0 \leq \gamma \leq 1$.

Au losange nous appliquons alors une convergence de boucle. Ceci repose sur un variant de boucle, c'est-à-dire une fonction monotone décroissante, qui satisfait pour toute valeur négative la post condition (cf. *infra*).

Modèle 2: Séquent de la vivacité atomique de l'action `takeoff`

$$
\begin{array}{l}
1\ \vdash (\\
2\ \quad p_x = 0 \wedge p_y = 0 \wedge p_z = 0 \wedge v_x = 0 \wedge v_y = 0 \wedge v_z = 0 \wedge \\
\quad\ \ a_x = 0 \wedge a_y = 0 \wedge a_z = 0 \wedge t = 0 \wedge dt = 1 \wedge \gamma = 0 \wedge \tau > 0 \\
3\ ) \mapsto \\
4\ [\\
\quad\ \ \texttt{\#initialisation de takeoff(10)} \\
5\ \quad \{ \\
6\ \quad\ \ alt := 10; dt := *; ?dt \geq 1 + 0.9 \times alt \wedge dt \leq 1 + 1.1 \times alt; vx := 0; vy := 0; \\
\quad\ \ \ a_x := 0; a_y := 0; a_z := 60dt^{-2}; \\
7\ \quad \} \\
\quad\ \ \texttt{\#takeoff - avancement} \\
8\ \quad \{ \\
9\ \quad\ \ t_0 := t; \{ \\
10\ \quad\ \ \ p'_x = v_x dt, y' = v_y dt, p'_z = 6\text{alt}\gamma(1-\gamma), \\
\quad\ \ \ \ v'_x = a_x dt, v'_y = a_y dt, t' = dt, \gamma' = 1, v'_z = a_z dt, a'_z = da_z \wedge \gamma \leq 1 \wedge t - t_0 < \tau \\
11\ \quad\ \ \} \\
12\ \quad \}^* \\
13\ ] \\
14\ \langle \\
\quad\ \ \texttt{\#takeoff - conclusion} \\
15\ \quad \{ \\
16\ \quad\ \ t_0 := t; \{ \\
17\ \quad\ \ \ p'_x = v_x dt, y' = v_y dt, p'_z = 6\text{alt}\gamma(1-\gamma), \\
\quad\ \ \ \ v'_x = a_x dt, v'_y = a_y dt, t' = dt, \gamma' = 1, v'_z = a_z dt, a'_z = -120dt^{-2} \\
\quad\ \ \ \ \wedge \gamma \leq 1 \wedge t - t_0 < \tau \\
18\ \quad\ \ \} \\
19\ \quad \}^* \\
20\ \rangle \\
21\ \gamma = 1
\end{array}
$$

$$\text{Convergence de boucle : } \frac{\Gamma \vdash \exists k.j(k),\ \Delta \qquad k \leq 0, j(k) \vdash \varphi \qquad k > 0, j(k) \vdash \langle a \rangle j(k-1)}{\Gamma \vdash \langle a^* \rangle \varphi,\ \Delta}$$

La fonction que nous introduisons à cet effet est $j(k) \equiv (\gamma = 1 - \frac{k\tau}{2dt} \vee \gamma = 1) \wedge 0 \leq \gamma \leq 1$. La première partie de cette fonction, $\gamma = 1 - \frac{k\tau}{2dt}$, contrôle la convergence quand γ est petit (ce qui revient à $k > 0$). En effet, sur une itération de boucle faisant passer de γ_- à γ_+, l'avance du système différentiel est limitée par la période du contrôleur, d'où $\gamma_+ - \gamma_- \leq \frac{\tau}{2dt} < \int_{\gamma_-}^{\gamma_+} \frac{\tau}{dt}$ quand $k > 1$, et $\gamma_+ - \gamma_- \leq \frac{k\tau}{2dt} < \int_{\gamma_-}^{\gamma_+} \frac{\tau}{dt'}$ quand $k \leq 1$.

La disjonction $\gamma = 1$, en conjonction avec $\gamma \leq 1$, empêche γ de croître au-delà de 1 lorsque k devient négatif. De cette clause, la formule finale est immédiate. □

La preuve du modèle ainsi présenté dépend faiblement de l'altitude, et l'on pourrait montrer plus généralement que, pour des conditions initiales similaires, le modèle est vrai pour toute altitude de décollage positive. Autrement, on pourra se contenter de réappliquer la preuve comme tactique pour prouver les décollages subséquents. Si l'on modifie à présent le modèle en introduisant une variable de ressource, la généralisation est évidemment invalidée. En conséquence, il sera souvent plus aisé de prouver pour un cas particulier et de réappliquer la preuve plutôt que de chercher les cas limites.

Le détail de la preuve a fait apparaître un invariant et un variant de boucle. Toutes les preuves de vivacité atomique se décomposent en l'exécution d'un préfixe $[\alpha]$, suivi de la complétion de l'action entrepise $\langle \omega \odot \bigwedge \phi_k \rangle$. Les actions comprenant des boucles non-déterministes, l'introduction de ces formules auxiliaires sera systématique.

- Pour l'induction, un invariant généralement satisfaisant sera l'état du système, exprimé en fonction de γ, avec $0 \leq \gamma \leq 1$.
- Pour la convergence, le variant dérivera des contraintes du domaine d'évolution, et l'on pourra commencer par

$$j(k) \equiv (\gamma = 1 - \frac{k\tau}{2\text{time}'} \vee \gamma = 1) \wedge 0 \leq \gamma \leq 1$$

Pour conclure sur la vivacité d'une action, intéressons-nous à présent à la vivacité séquentielle de l'action `takeoff`. À la suite du décollage, le système entreprend de rejoindre soit le premier pylône, soit la zone d'atterrissage d'urgence. Le choix de la destination est donné par le niveau de charge de la batterie, la valeur seuil étant de 30%. Dans les deux cas, l'action effectuée est un `goto`, dont les prérequis sont de ne plus être au sol. Nous pouvons donc simplifier ce prérequis : $bat > 30 \wedge p_z > 0 \vee bat \leq 30 \wedge p_z > 0 \leftrightarrow p_z > 0$. Pour cette démonstration, nous introduisons un modèle complet (donc avec la batterie) conforme à la spécification `Sophrosyne` du système donnée au paragraphe 4.4. Ici encore, nous utilisons les résultats de notre solveur (paragraphe 6.3.1) pour ajuster le modèle à la syntaxe et aux compétences de `KeYmaera X`.

Théorème : vivacité séquentielle de `takeoff`. *L'action `takeoff` de la mission d'inspection satisfait la propriété de vivacité séquentielle : le séquent présenté dans le modèle 3 est satisfait.*

Démonstration. Ici encore nous n'explicitons pas la preuve. Seul l'invariant de boucle (et l'ordre de preuve de ses différentes clauses) nécessiterait de l'astuce s'il ne nous était pas directement donné par le solveur (voir 6.8). □

8.2 La sûreté

Une mission satisfaisant la propriété de vivacité est correctement définie pour son exécution. `Sophrosyne` propose de plus d'intégrer de la supervision dans les missions, en utilisant les

Modèle 3: Séquent de la vivacité séquentielle de l'action `takeoff`

$\vdash$ (
  $p_x = 0 \wedge p_y = 0 \wedge p_z = 0 \wedge v_x = 0 \wedge v_y = 0 \wedge v_z = 0 \wedge$
  $a_x = 0 \wedge a_y = 0 \wedge a_z = 0 \wedge t = 0 \wedge dt = 1 \wedge \text{bat} = 100 \wedge \gamma = 0 \wedge \tau > 0$
) $\mapsto$
[
  `#initialisation de takeoff(10)`
  {
    $alt := 10; dt := *; ?dt \geq 1 + 0.9 \times alt \wedge dt \leq 1 + 1.1 \times alt; vx := 0; vy := 0;$
    $a_x := 0; a_y := 0; a_z := 60dt^{-2};$
  }
  `#takeoff`
  {
    $t_0 := t;$ {
      $p'_x = v_x dt, y' = v_y dt, p'_z = 6\text{alt}\gamma(1-\gamma),$
      $v'_x = a_x dt, v'_y = a_y dt, v'_z = a_z dt, a'_z = -120/dt^2, t' = dt, \gamma' = 1,$
      $bat' = -0.004 * (9.89 + a_z) * dt \wedge \gamma \leq 1 \wedge t - t_0 < \tau \wedge bat \geq 0$
    }
  }*
  $?\gamma = 1;$ `#takeoff terminé`
]
$p_z > 0$

contextes. Dans ce processus de définition de mission, l'opérateur espère naturellement que certaines propriétés perduront tout du long de la mission, par exemple que les batteries ne seront pas vides avant que le drone n'ait atterri. Et, s'il a pu introduire un contexte englobant toute la mission dont l'effet est de faire atterrir d'urgence le drone quand le niveau de charge atteint un seuil critique, il reste souhaitable de ne pas avoir à recourir à de tels moyens ; il s'agit en quelque sorte d'une ceinture de sécurité, mise avec l'espoir qu'elle ne sera pas nécessaire. C'est précisément cet aspect sécuritaire que recouvre la propriété de sûreté, qui qualifie un contexte.

Remarquons toutefois que cette propriété nécessite l'intervention de l'opérateur : tous les contextes n'ont pas vocation à être sûrs, certains peuvent introduire un branchement souhaité dans le cadre d'une exécution nominale. `Sophrosyne` ne comporte pas de syntaxe discriminante selon que le contexte doit être sûr ou non, car en pratique, les deux catégories sont traitées de la même manière à l'exécution.

Sûreté d'une mission. *Une mission est dite sûre lorsque pour toute action ω exécutée dans des contextes ϕ_k, $k = 0..n$ parmi lesquels les contextes aux gardes ϕ_i, $i \in \mathcal{I} \subset \{0..n\}$ sont jugés critiques, depuis un état ψ, on a :*

$$\psi \vdash \left[\omega \odot \bigwedge_{k=0..n} \phi_k\right] \left\langle \omega \odot \bigwedge_{k=0..n} \phi_k \right\rangle \left(\bigwedge_{i \in \mathcal{I}} \phi_i \wedge (\gamma = 1 \vee \bigvee_{j \notin \mathcal{I}} \neg \phi_j) \right)$$

Remarque : *en pratique, lorsqu'un contexte abortif ϕ^β est critique, nous pouvons réécrire cette formule en vérifiant que la porte associée δ est restée ouverte.*

$$\psi \wedge \delta \vdash \left[\phi^\beta(\omega);\ ?\gamma = 1;\right] \delta$$

8.2.1 Un exemple de sûreté

Pour illustrer la vérification de la sûreté, nous allons considérer la supervision de la charge de la batterie dans le cadre de la mission fil rouge. Toutefois, nous l'avons vu, ces preuves deviennent rapidement fastidieuses du fait de la complexité des modèles des actions. Nous allons donc les simplifier en en conservant la substance afin de nous concentrer sur la notion de sûreté qui nous intéresse ici. En particulier nous ôtons la composante de décharge de la batterie liée à la vitesse horizontale en l'absence d'expression directe des fonctions racines dans l'assistant de preuve.

Théorème : sûreté du contexte de la charge de la batterie. *Le contexte de charge de la batterie de la mission d'inspection satisfait la propriété de sûreté.*

Démonstration. Nous ne développons pas le contexte sur toute la mission, nous nous contentons de donner un exemple de preuve associée, celle concernant la première action `goto` permettant au système de rejoindre le premier pylône. Le séquent correspondant est présenté dans le modèle 4.

Encore une fois, nous n'écrivons ici que les invariants de boucle qui sont nécessaires, et qui nous sont donnnés par notre solveur. Le premier invariant concerne l'action `takeoff`, nous l'avons déjà rencontré à deux reprises et ne revenons pas dessus (cf paragraphe 6.3.1). Le second invariant est celui de l'action `goto`, proposé ci-dessous

$$\begin{aligned} \gamma \geq 0 \ \wedge\ \gamma \leq 1 \ \wedge\ \mathrm{gate}_0 = 0 \wedge \\ p_x = 0 \ \wedge\ d_{x1} = 0 \ \wedge\ d_{y1} = -5 \wedge \\ p_y = 10 \times \gamma^3 - 15 \times \gamma^2 \ \wedge\ d_{z1} = 0 \wedge \\ p_z = 10 \ \wedge\ \mathrm{bat} \geq 99.52528 - 0.004 \times 9.89 \times 5 \times \gamma \\ \wedge\ \tau > 0 \end{aligned}$$

Cet invariant donne l'évolution de l'état du système lors de l'exécution de l'action `goto`. Pour borner la charge de la batterie, il nous faut considérer le pire cas lors de l'exécution de l'action `takeoff`. Celui-ci est obtenu quand la dérivée du temps vaut 12 ($99.2528 = 100 - 0.004 \times 12 \times 9.89$). □

8.3 L'incertitude liée au modèle

Le paragraphe précédent a montré comment de considérations de sûreté de fonctionnement procèdent des designs de missions n'invalidant pas les gardes correspondantes. Ces garanties, nous l'avons vu au chapitre 3 ne concernent toutefois que quelques modèles, dans notre cas quelques exécutions symboliques. L'exécution réelle, elle, est susceptible de mener le système dans des états dangereux qui invalideraient ces gardes. Les missions de repli associées se déclencheraient alors, donc elles devraient répondre aux mêmes exigences de correction que nous avons décrites précédemment : vivacité et sûreté. Cette vérification repose par exemple sur la clauses 8.6 de la vivacité atomique qui, par définition de la propriété de sûreté, n'est plus atteinte. Vérifier les missions de repli de contextes sûrs nécessite donc préalablement de construire de nouveaux états initiaux faisables, c'est-à-dire qui invalident la garde concernées.

L'ensemble initial le plus général résulterait de la négation de la garde, considérant par exemple tous les états pour lequels le niveau de charge de la batterie est inférieure à un seuil

Modèle 4: Séquent de sûreté du contexte de la batterie pour la première action `goto`

$$
\begin{array}{l}
\vdash (\\
\quad p_x = 0 \wedge p_y = 0 \wedge p_z = 0 \wedge t = 0 \wedge dt = 1 \wedge \mathrm{bat} = 100 \wedge \gamma = 0 \wedge \tau > 0 \wedge \mathrm{gate}_0 = 0 \\
) \mapsto \\
[\\
\quad \{ \\
\qquad \texttt{\#initialisation de takeoff(10)} \\
\qquad \mathrm{alt} := 10; dt := *; ?dt >= 1 + 0.9 \times \mathrm{alt} \wedge dt \leq 1 + 1.1 \times \mathrm{alt}; \\
\quad \} \\
\quad \{ \\
\qquad t_0 := t; \qquad \texttt{\#takeoff} \\
\qquad \{ \\
\qquad\quad p'_x = 0, p'_y = 0, p'_z = 6 \times \mathrm{alt} \times \gamma \times (1 - \gamma), \\
\qquad\quad \mathrm{bat}' = -0.004 \times (9.89 \times dt + 6 \times \mathrm{alt} \times (1 - 2 \times \gamma)/dt),\ t' = dt, \gamma' = 1 \\
\qquad\quad \wedge t - t_0 < \tau \wedge \gamma \leq 1 \wedge \mathrm{bat} \geq 0 \\
\qquad \} \\
\quad \}^* \\
\quad \{?\gamma = 1; \gamma := 0;\} \qquad \texttt{\#takeoff terminé} \\
\quad \{ \\
\qquad ?\mathrm{gate}_0 = 0; \qquad \texttt{\#porte du contexte ouverte} \\
\qquad \{ \\
\qquad\quad \texttt{\#initialisation goto(0, -5, 10)} \\
\qquad\quad ?\mathrm{bat} > 30; d_{x1} := 0 - p_x; d_{y1} := -5 - p_y; d_{z1} := 10 - p_z; \\
\qquad \} \\
\qquad \cup \{ \\
\qquad\quad ?\neg\mathrm{bat} > 30; \gamma := 0; \mathrm{gate}_0 := 1; \ldots \qquad \texttt{\#mission de repli} \\
\qquad \} \\
\quad \} \\
\quad \cup \{?\mathrm{gate}_0 = 1;\} \qquad \texttt{\#porte du contexte fermée} \\
\quad \{ \\
\qquad \{ \\
\qquad\quad ?\mathrm{gate}_0 = 0; \qquad \texttt{\#porte du contexte ouverte} \\
\qquad\quad \{ \\
\qquad\qquad ?\mathrm{bat} > 30; t_0 := t; dt := *; ?dt > 0 \wedge dt^2 = d_{x1}^2 + d_{y1}^2 + d_{z1}^2; \qquad \texttt{\#goto} \\
\qquad\qquad \{ \\
\qquad\qquad\quad p'_x = 6 \times d_{x1} \times \gamma \times (1-\gamma), p'_y = 6 \times d_{y1} \times \gamma \times (1-\gamma), p'_z = 6 \times d_{z1} \times \gamma \times (1-\gamma), \\
\qquad\qquad\quad \mathrm{bat}' = -0.004 \times (9.89 \times dt + 6 \times d_{z1} \times (1 - 2 \times \gamma)/dt), \\
\qquad\qquad\quad t' = dt, \gamma' = 1 \wedge t - t_0 < \tau \wedge \gamma \leq 1 \wedge \mathrm{bat} \geq 0 \\
\qquad\qquad \} \\
\qquad\quad \} \\
\qquad\quad \cup \{ \\
\qquad\qquad ?\neg\mathrm{bat} > 30; \gamma := 0; \mathrm{gate}_0 := 1; \ldots \qquad \texttt{\#mission de repli} \\
\qquad\quad \} \\
\qquad \} \\
\qquad \cup \{?\mathrm{gate}_0 = 1;\} \\
\quad \}^* \\
\quad \{?\gamma = 1;\} \qquad \texttt{\#goto ou repli terminé} \\
] \\
\mathrm{gate}_0 = 0
\end{array}
$$

critique. Ce type de condition est évidemment inadapté : nous sommes certains d'y trouver des états desquels procèdera une invalidité des missions résultantes. On s'en convaincra facilement en considérant une chute instantanée de la charge de la batterie à 0%, ou en songeant à la téléportation du système en un lieu lointain. De fait nous ne serons jamais en mesure de prouver quoi que soit en choisissant de vastes ensembles initiaux car ils deviennent immédiatement irréalistes ou suffisamment peu probables pour suggérer, s'ils surviennent, des problèmes bien plus inquiétants que ceux provenant de la définition de mission (*e.g.* défaillance de capteur critique, environnement hostile). Toutefois notre objectif est d'assurer que la mission est suffisamment résiliente pour supporter de petits défauts transitoires, tels que des bourrasques de vent ou du bruit sur les données de capteur. À cet effet, les états initiaux que nous envisageons sont calculés en introduisant des perturbations sur l'état. Ces perturbations peuvent être la modélisation de phénomènes physiques, par exemple un modèle de turbulence de Dryden, ou un modèle générique qui recouvrent diverses situations, en considérant une évolution non-déterministe bornée dans l'espace des états. L'introduction de ces éléments constitue une perspective concrète de notre travail, et des développements sont encore nécessaires pour assurer leur correcte intégration avec la production automatique des obligations de preuves.

8.4 Conclusion et perspectives

Ce chapitre a introduit une notion de régularité de la mission comprenant deux volets, le premier d'exécutabilité adossé sur la vivacité, et le second de sûreté. Nous avons montré comment les obligations de preuves afférentes étaient dérivées, et avons étendu la notion de sûreté à une quantification par perturbation.

Tout ceci dote donc `Sophrosyne` d'un cadre théorique pour analyser ses missions, et il reste à affiner sa mise en pratique. En l'état, les outils auxquels nous avons recours pour l'étude formelle des missions sont :

- le générateur d'obligations de preuves de `sophrosyne`
- le solveur de `symsophrosyne`
- l'assistant de preuve `KeYmaera X`

Nous voyons deux principales limites à notre solution, l'intégration et la spécialisation. La première se traduit par des interactions encore trop sommaires, nécessitant une intervention humaine à bien des endroits. Les résultats que nous obtenons par le solveur de `symsophrosyne`, en particulier la caractérisation de chaque variable d'état, interviennent régulièrement dans la preuve. Leur report n'est toutefois pas automatique, et nous ne capitalisons que rarement sur les connaissances que nous obtenons sur les systèmes différentiels, ce qui se traduit par de nombreuses répétitions lors de la preuve. Au sein de l'intégration nous considérons en effet les problématiques associées à la gestion de preuve : enregistrer des preuves, des sections de preuve comme lemme, rejouer des preuves, réutiliser des lemmes, récapituler les preuves et leur avancement. Ceci peut également être vu comme un des aspects de la seconde limite : le manque de spécialisation des outils. Cette limite concerne toutefois principalement l'assistant de preuve. Les contraintes auxquelles nous faisions face (*e.g.* temps de développement) nous ont fait choisir la seule solution préexistante `KeYmaera X`. Sans lui apporter de modifications, cette version [18] montre toutefois des limites pour notre application. Tout d'abord, des fragments de la logique dynamique différentielle dont nous avons un usage récurrent ne sont pas implémentés, notamment les inégalités différentielles. La logique dynamique différentielle prévoit en effet de les intégrer par un traitement analogue à celui que nous en faisons, c'est-à-dire par une quantification [150] :

18. ce passage a été écrit en référence à la dernière version en date de `KeYmaera X`, à savoir la 4.9.1

$\exists u.\, x' = f(x) + u \land u \geq 0$. Ces écritures ne sont pas reconnues par l'analyseur syntaxique de `KeYmaera X`, ce qui doit nous faire envisager sa modification ou une autre implémentation d'un assistant de preuve pour la logique dynamique différentielle. Le manque de spécialisation intervient également au niveau des règles du calcul des séquents, en particulier sur le traitement de l'opérateur $\langle . \rangle$. En guise d'exemple, nous sommes régulièrement amenés à considérer un séquent de la forme $P \land \gamma = 1 \vdash \langle x' = f(x) \land Q \rangle \gamma = 1$ pour l'analyse de la vivacité d'une mission. La sémantique de la logique dynamique différentielle autorise une règle d'état initial des évolution différentielles que nous écrivons

$$\frac{P \vdash Q \land R}{P \vdash \langle x' = f(x) \land Q \rangle R}$$

permettant de remplacer la vérification d'une propriété après une évolution différentielle par le cas particulier de son état initiale. Le mérite de cette règle est d'éviter la résolution du système différentiel, non seulement coûteuse en terme de performances mais aussi parfois impossible. Ce dernier point laisse d'ailleurs penser que cette règle doit être érigée comme axiome car ne pouvant être démontrée dans le cas général à partir des autres règles. Une telle règle est en pratique implémentée dans `KeYmaera X` sous le nom de « diamond differential skip », cependant celle-ci n'est pas référencée rendant son utilisation impossible sans appel manuel. Cela corrobore l'idée que la preuve formelle de mission `Sophrosyne` nécessite une utilisation de la logique dynamique différentielle et de `KeYmaera X` s'écartant leurs motivations premières et qu'elle bénéficierait d'un assistant de preuve dédié.

Chapitre 9

L'application de planification de missions

La forme, c'est le fond qui remonte à la surface

– Hugo

Les chapitres précédents ont introduit le coeur de la méthode et des outils `Sophrosyne`. Ils permettent de programmer une mission, de l'analyser avec une notion formelle de correction, et de l'exécuter dans le cadre d'une opération avec le système cyber-physique réel. Cet usage demande de la part de l'utilisateur une expertise en la matière : bien que `Sophrosyne` reste en premier lieu un langage dédié avec un noyau très restreint, il suppose des connaissances en analyse (*e.g.* systèmes différentiels), en programmation (*e.g.* continuations), ou encore en logique (*e.g.* preuve formelle) pour exprimer son plein potentiel. Afin d'amoindrir cette barrière à l'entrée, divers outils et méthodes complémentaires peuvent être envisagés, ce que nous allons illustrer dans ce chapitre. Une interface homme-machine est un composant incontournable pour assister l'utilisateur, et nous avons développé une application de planification de mission qui répond à plusieurs enjeux :

- segmenter les rôles et étendre la classe d'utilisateurs à des populations profanes en programmation
- assister la conception de missions `Sophrosyne`, notamment à l'aide d'outils de visualisation et de simulation
- représenter graphiquement la trace d'exécution d'une mission `Sophrosyne`

Du point de vue du cycle de vie d'une mission (planification, vérification, exécution, et revue), cette application renforce en particulier les étapes extrêmes, qui se résument auparavant à d'un côté un exercice de programmation, de l'autre des journaux d'exécutions produits par la bibliothèque d'exécution. Ces besoins se sont trouvés renforcés par l'application de `Sophrosyne` à un projet d'inspection d'infrastructures par drone. Dans ce cadre applicatif émergent les problématiques sus-mentionnées, d'assistance à la planification de mission et de production d'un rapport d'opération.

Après une succincte exposition du projet dont nous tirerons des illustrations pour la suite, nous présenterons l'application de planification de mission par les différents rôles que peuvent endosser les utilisateurs. Chaque rôle requiert un ensemble de fonctionnalités, et leur énumération n'est pas l'objet de cet ouvrage. Nous verrons toutefois celles qui étendent directement `Sophrosyne` pour assister la conception de mission, que ce soit pour leur vérification au moyen

de simulations, ou pour une expression de programmes `Sophrosyne` les interfaçant avec les technologies web et avec les fonctions de visualisation prévues.

9.1 Le projet `ceos`

Le projet `ceos` [18] ambitionne de proposer une solution de mini-drone professionnel pour l'inspection d'ouvrages d'Opérateurs d'Importance Vitale. Pour cela, trois cas d'usage ont été identifiés :

- l'inspection de clôtures aéroportuaires pour l'aéroport de Caen
- l'inspection de conduites forcées pour EDF
- l'inspection de lignes moyenne tension pour Enedis

Ces applications sont critiques pour différentes raisons : risque aérien du fait de la proximité d'avions, conditions environnementales rudes en montagne, zones à niveau de densité de population non négligeable. À cela s'ajoute des vols hors vue (similaire à un scénario S4, mais avec un drone de 25 kg voir chapitre 1) et un fort degré d'automatisation. La criticité est en conséquence un enjeu central du projet, qui s'est décliné par la modification d'un autopilote open-source en une version temps réel, l'utilisation d'un système d'exploitation temps réel, la redondance des communications, et l'adjonction de « couloirs intelligents » pour la navigation. La conception de ce dernier composant, qui nous incombe, fut l'occasion d'éprouver `Sophrosyne` pour la définition de missions. La plateforme drone est illustrée dans les figures 9.1 et 9.2.

FIGURE 9.1 – Drone `ceos`

FIGURE 9.2 – Expérimentation terrain : inspection de lignes électriques en Lorraine

Notre contribution dans `ceos` se situe sur quatre plans :

1. la gestion de projet pour l'Université de Lorraine
2. l'établissement d'une plateforme de simulation pour assister la conception des composants des partenaires techniques du projet
3. la conception d'une interface homme-machine pour la planification de la navigation
4. la conception d'un composant logiciel embarqué assurant la navigation automatique

Nous ne détaillerons ici aucun de ces éléments : notre propos n'est pas tant la présentation du projet `ceos` que la démonstration de l'utilisation de `Sophrosyne` au sein d'un projet complexe. Une plateforme de simulation plus complète sera évoquée au paragraphe 9.3.1. De même, l'objet de ce chapitre est de présenter une application de planification qui dépasse le périmètre du projet `ceos`. Elle en partage toutefois l'interface, que nous présentons dans la figure 9.3. Des informations supplémentaires sur la navigation automatique sont données dans l'annexe C. Soulignons par ailleurs l'absence de vérification formelle de la mission : celle-ci n'était pas une exigence du

projet, ce qui nous a conduit à prioriser l'intégration d'autres composants dans l'application de planification de missions `Sophrosyne`.

FIGURE 9.3 – L'application de planification de mission déclinée pour le projet `ceos`

9.2 La diversification des utilisateurs

La diversification des utilisateurs est elle-même source de nouvelles contraintes en ce qu'elle appelle à une réflexion sur les différents rôles qu'ils endossent. La division résultante réplique la segmentation introduite par `Sophrosyne`, en même temps qu'elle la complète avec des rôles associés aux fonctionnalités propres à l'interface graphique et son usage. Les rôles hérités de `Sophrosyne` sont

- l'**ingénieur système**, qui définit un système cyber-physique par ses spécifications
- le **planificateur**, qui écrit une mission pour un système donné
- l'**ingénieur méthodes formelles**, qui vérifie formellement la correction d'une mission

À ceux-ci s'ajoutent les rôles

- d'**opérateur**, qui supervise et endosse la responsabilité de l'exécution réelle de la mission (comprend le télé-pilotage)
- de **cartographe**, qui produit des cartes relatives à la mission (*e.g.* objets d'intérêt, contraintes réglementaires)

ainsi que ceux auxiliaires

- de **gestionnaire** pour la gestion des utilisateurs et l'affectation de rôles
- d'**invité** pour observer sans contribuer au cycle de vie de la mission

Une même personne peut naturellement remplir plusieurs rôles ; en particulier, l'ingénieur méthodes formelles est susceptible d'officier comme planificateur également. Ces deux rôles ne doivent toutefois pas être confondus : la vérification formelle de mission est encouragée mais son absence n'est pas prohibitive à la tenue d'une opération. De même, nous distinguons la définition

de système et la vérification formelle de missions afin de ne pas introduire de collusion entre ces rôles, le risque étant la modification d'un système pour satisfaire les besoins d'une preuve.

Une conséquence technique de cette séparation des rôles est d'introduire un mécanisme d'authentification et d'autorisation, offrant une expérience dédiée à chaque profil. L'authentification s'effectue par échange de jetons (JSON Web Token). Le partage de ressources entre les différents utilisateurs et la définition des droits afférents rend l'implémentation et la maintenance des autorisations délicates. Nous avons donc opté pour un service de contrôle d'accès dédié : `Open Policy Agent` (*OPA*) [14]. Celui-ci se comporte comme un système de contrôle d'accès basé sur attributs. Quand un utilisateur veut accéder à une ressource, le serveur envoie une requête à l'*API REST* d'*OPA* qui évalue la requête selon des politiques prédéfinies écrites en `Rego`, un langage dédié. Le serveur doit fournir les informations nécessaires à l'évaluation de la requête, par exemple le détenteur de la ressource. Si toutes les informations sont fournies, *OPA* renvoie une valeur de vérité signifiant la décision. Sinon, le centre d'évaluation des politiques peut renvoyer un filtre, qui, une fois converti en une requête `SQL`, permet de filtrer les éléments sur lesquels l'utilisateur a le droit d'agir. Cette dernière utilisation améliore les performances en évitant le transit de grandes quantités de données sur le réseau.

La présentation des fonctionnalités de l'application de planification de mission qui suit reprend chacun de ces rôles. Nous ne développerons pas les rôles auxiliaires de gestionnaire et d'invité.

9.2.1 L'ingénieur système

La définition d'un nouveau système est en l'état une tâche peu assistée. L'ingénieur système n'a à sa disposition qu'une interface guidant le remplissage de chaque champ du système (voir figures 9.4 et 9.5).

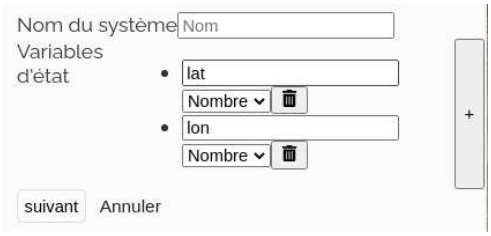


FIGURE 9.4 – Définition d'un système : déclaration du nom et des variables d'état

Définir un système requiert un nom, des variables d'états et les actions et missions exécutables. Le renseignement des éléments cartographiques et de l'état final des actions et missions nécessite généralement des calculs relatifs à l'état du système au départ de l'action et aux paramètres de l'action, qui s'effectuent au moyen d'un langage dédié minimal que nous présentons au paragraphe 9.3.2

À terme, tous ces éléments devraient pouvoir être déclarés à partir d'un fichier de système `Sophrosyne`, d'un fichier de mission `Sophrosyne`, et de leur analyse par le solveur pour récupérer les éléments cartographiques et l'état final associés à chaque action et mission. En plus d'automatiser le processus de définition d'un nouveau système, cette fonctionnalité permettrait de garantir la cohérence entre la représentation graphique d'un calque de mission et les états accessibles.

9.2.2 Le planificateur

Afin de permettre à des utilisateurs non initiés à la programmation de définir des missions, le rôle de planificateur de missions peut s'appuyer sur une bibliothèque de missions préprogrammées. Nous avons eu recours à cette technique dans le cadre du projet `ceos` (le composant « Navigation `Ceos` » présenté dans la figure C.1), les exigences décrivant parmi les utilisateurs des individus ayant une faible maîtrise de l'outil informatique. Seules des compositions séquentielles

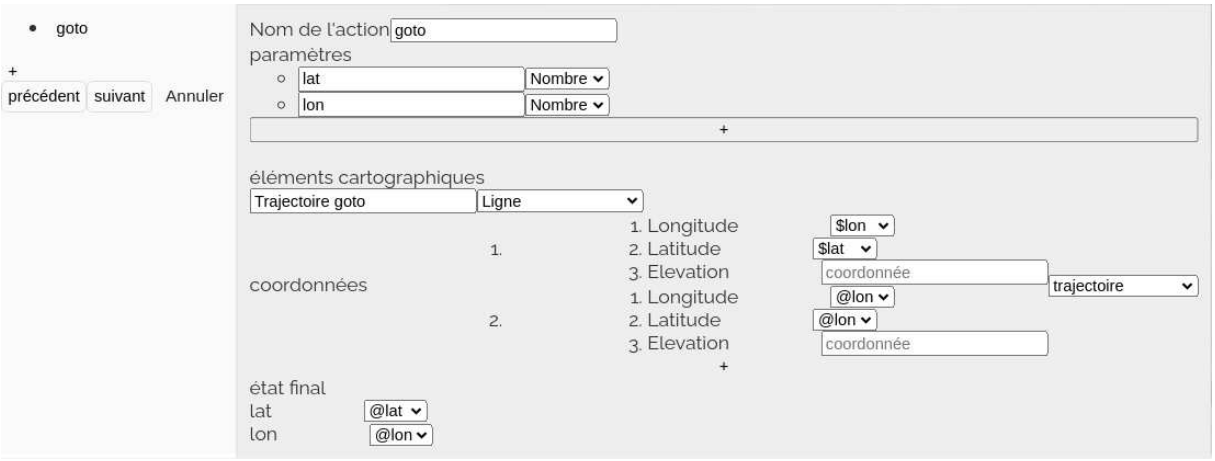


FIGURE 9.5 – Définition d'un système : déclaration d'une action ou d'une mission

d'actions et missions sont nécessaires pour définir une mission `ceos`, évitant au planificateur toute gestion de branchement dans l'exécution. Un exemple d'utilisation de l'interface de définition de mission pour une opération d'inspection est présentée dans la figure 9.6.

9.2.3 L'ingénieur méthodes formelles

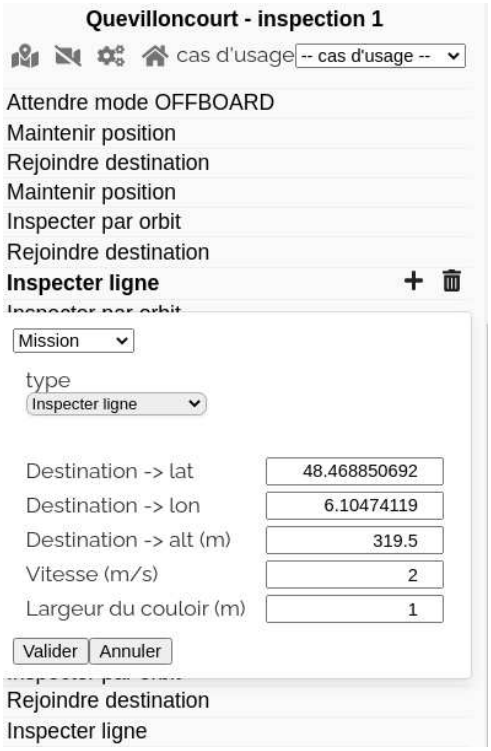


FIGURE 9.6 – Interface de définition de mission

L'ingénieur méthodes formelles s'intéresse à la correction d'une mission vis-à-vis du modèle de système. Pour cela, les obligations de preuves de vivacité et sûreté sont automatiquement générées. Le développement d'un assistant de preuve formelle dépasse le cadre de ce travail de thèse, et nous avons opté pour `KeYmaera X` pour remplir cette fonction (cf chapitre 8). De plus, la preuve formelle de mission n'étant pas retenue comme technique de vérification dans le projet `ceos`, l'ensemble des fonctionnalités afférentes ont été laissée à des itérations ultérieures de l'application. L'interaction entre l'ingénieur méthodes formelles et l'interface graphique que nous présentons dans ces lignes en est d'autant réduite.

9.2.4 L'opérateur

Diverses fonctionnalités offrent à l'opérateur des repères avant d'exécuter la mission avec le système réel. En premier lieu, le fond de carte, les calques de données, et le calque de mission dessinent le cadre géographique d'exécution. Ces informations sur l'environnement

peuvent être complétées par *e.g.* des prévisions et estimations météorologiques que nous récupérons de l'*API* `OpenWeatherMap` [3]. Pour plus de précision sur la trajectoire qui sera entreprise par le véhicule, l'opérateur dispose du rendu graphique de la simulation. Quand il est prêt pour exécuter la mission, il peut extraire des *geofences* à charger dans l'autopilote, et récupérer la mission exécutable.

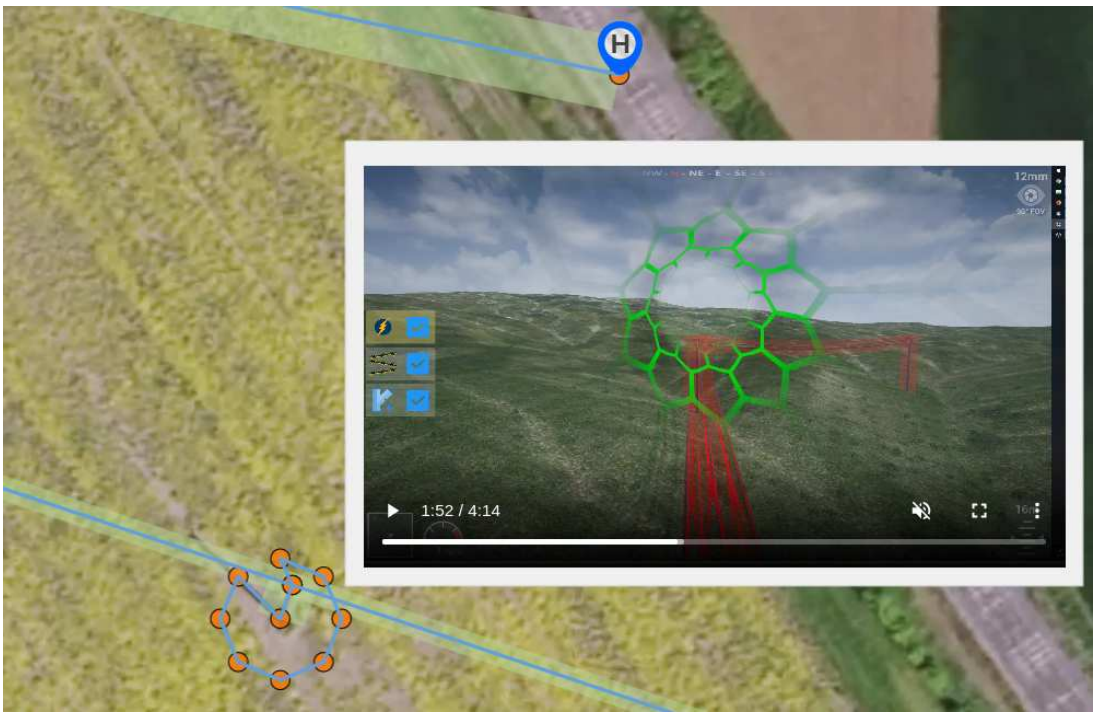


FIGURE 9.7 – Rendu graphique d'une simulation

9.2.5 Le cartographe

Le cartographe est un deuxième rôle attenant au processus de programmation des systèmes cyber-physiques mobiles avec `Sophrosyne`. Sa fonction est de produire des calques introduisant des données environnementales utiles à la définition de mission, et à son exécution par l'opérateur. Ils contiennent par exemple des objets d'intérêt, à l'instar des pylônes électriques à inspecter, des *geofences* réglementaires ou dédiées à la mission, ou des éléments d'attention tels que des délimitations de zones peuplées ou des obstacles statiques (voir figure 9.8). En plus de cette valeur informative, les calques de données s'articulent de deux manières avec `Sophrosyne`. La première permet de générer automatiquement une mission à partir d'un calque. Cette fonction se limite pour l'instant à quelques exemples spécifiques au projet céos. Ainsi, à partir d'un calque du réseau de transport d'électricité nous déduisons l'essentiel d'une mission d'inspection comprenant les inspections linéaires de lignes électriques et circulaires autour des pylônes. La deuxième articulation se fait avec les composants de simulation. Ces calques de données servent à générer des mondes représentatifs de la zone dans laquelle s'effectuera l'opération, renforçant l'intérêt du moteur de rendu graphique.

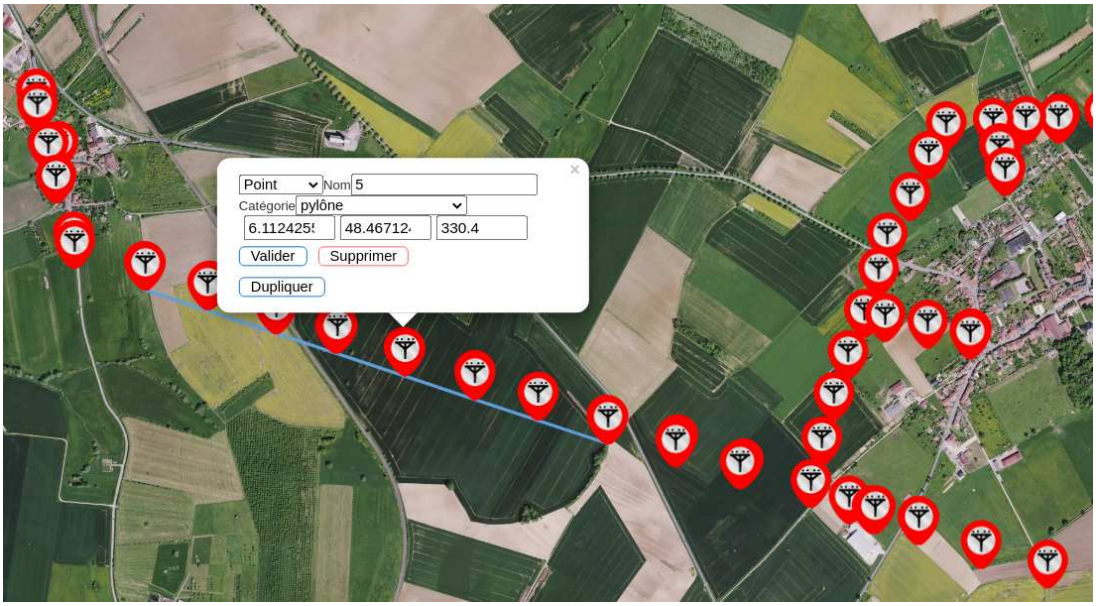


FIGURE 9.8 – Exemple de calque de données : pylônes et portion de lignes électriques

9.3 L'assistance à la conception de missions

9.3.1 Une démarche de (co-)simulation

La conception de missions fait intervenir différents acteurs et diverses interactions en résultent. Dans une démarche agile, l'équipe répondant au besoin formulé par le client tire profit de rendus intermédiaires qui supportent la discussion avec le client. Ils permettent de suivre l'évolution du projet et de veiller à ce qu'il emprunte la voie attendue. Les coûts engagés et le temps consommé, par exemple pour l'obtention de laissez-passer dans le cadre d'une opération drone, s'accommodent difficilement des nombreux soucis de dernière minute, tels que des prises de vues inadéquates lors d'une inspection. Ce problème d'adéquation entre la demande client et la réponse apportée n'est certes pas spécifique aux applications robotiques, et pour contrecarrer cela, la simulation est une stratégie communément adoptée.

À cet effet, la chaîne outillée que nous proposons intègre une plateforme de simulation, que nous avons mises en oeuvre autour de versions de l'autopilote PX4. Ces différentes simulations s'inscrivent également dans une démarche de conception incrémentale, chaque étape rapprochant du cadre opératoire final. Quatre catégories de simulation sont considérées dans le développement basé sur les modèles :

- *Model In the Loop* (MIL), ne comprenant qu'un modèle du contrôleur
- *Software In the Loop* (SIL), intégrant le logiciel final du contrôleur
- *Processor In the Loop* (PIL), exécutant le logiciel sur la carte électronique finale
- *Hardware In the Loop* (HIL), associant également des capteurs ou actuateurs du système réel [19]

Le développement de missions `Sophrosyne` s'inscrit librement dans ce cadre basé sur les modèles, ce que résume le tableau 9.1. La première étape (MIL) vérifie formellement la mission `Sophrosyne` dans un cadre analytique. De plus, il n'est pas incongru de voir la production

19. Il est à noter que les simulations HIL est PIL sont occasionnellement confondues, les développeurs PX4 parlant d'HIL pour ce qui est communément appelé PIL.

d'un calque décrivant la mission comme une première simulation MIL même simpliste : selon la précision et l'exhaustivité des éléments cartographiques dérivés, le calque de mission projette l'évolution de l'état sur un système d'information géographique. Le code `Python` généré est alors testé dans un environnement dédié, pour PX4 dans une simulation (SIL) portée par un simulateur robotique tel que `Gazebo` [9] et comprenant le code final de l'autopilote. Puis, la même simulation est effectuée en déployant le code sur son environnement d'exécution matériel (PIL). Enfin, d'autres composants matériels sont intégrés à la simulation (HIL), à l'instar d'une carte Pixhawk [15] supportant l'autopilote.

Table 9.1 – Récapitulatif des simulations entreprises et catégorisation

Simulation	Catégorie	Contrôleur	Système
Calque de mission	MIL	Mission `json`	Système `json`
Partition de mission	MIL	Mission `sophrosyne`	Système `sophrosyne`
Logicielle	SIL	Mission `python`	Système `python` + simulateur physique
Matérielle	PIL	Mission `python` sur matériel final	Idem
Matérielle 2	HIL	Idem	Idem avec éléments de la plateforme matérielle (autopilote, capteurs)

Ces différentes étapes fournissent des jalons pour structurer l'avancement du projet, et différents rendus y sont associés. Le travail formel sur les modèles (MIL) dispose ainsi des partitions de mission, utiles pour mener des discussions techniques et conduire l'analyse de risques. L'interaction avec les collaborateurs non techniques reposera davantage sur des rendus graphiques 3D de la mission, apparaissant à partir de la simulation SIL.

Mais l'utilité d'une visualisation 3D réaliste de la mission dépasse la seule communication. Les drones embarquent systématiquement des caméras, et l'automatisation des missions suppose en particulier l'automatisation du traitement de ces flux, tâche jusqu'alors effectuée par le télé-pilote. Le rendu photo-réaliste permet d'étendre la portée de la simulation, en intégrant et validant les algorithmes de traitement d'images qui sont embarqués sur le système réel. Les avantages de cette méthode sont multiples : en plus de diminuer les coûts de la validation, elle permet de travailler sur des images souvent difficilement obtenables pour des raisons techniques et réglementaires. Dès lors, l'architecture que nous avons mise en place comprend un module séparé dédié à la visualisation, si bien qu'il est adéquat de parler de co-simulation (voir figure 9.9). Le rendu graphique 3D est délégué à `Unreal Engine 4` [22], un moteur graphique pour jeu vidéo constituant l'état de l'art dans le domaine. Pour cela, un monde est généré à partir des données de calques envoyées au moteur de rendu graphique. À l'exécution de la simulation, les flux télémétriques sont transmis au moteur graphique, permettant de faire évoluer le point de vue de l'observateur. Le flux vidéo de l'observateur est alors partagé au module de traitement d'image, et diffusé dans l'application web.

9.3.2 `Sophrosyne` version web

Pour l'interaction avec l'interface web des données de systèmes et missions doivent être échangées entre le serveur et le client. Bien qu'il soit possible de communiquer des fichiers `sophrosyne`, cela supposerait d'effectuer une analyse lexicale et syntaxique des deux côtés (client et serveur) avant de les utiliser. Nous optons pour une représentation dédiée à l'interface graphique, au

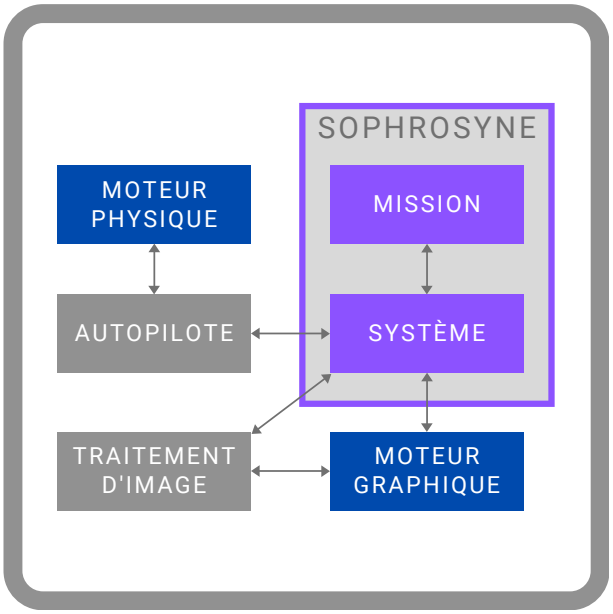


FIGURE 9.9 – Architecture de co-simulation SIL, PIL et HIL pour les missions `Sophrosyne` : en bleu les simulateurs, en gris les modules externes. Les modules `Sophrosyne` sont du code exécutable (`python`) en partie généré

format `json`, dont un résumé est présenté par l'exemple dans les figures 9.10 et 9.12. Nous en profitons pour simplifier la représentation des actions : l'interface web n'a pas vocation à conduire des raisonnements sur le modèle analytique, et nous remplaçons les systèmes différentiels par :

- une liste d'éléments cartographiques associés
- un calcul de l'état final

Nous écrivons « l'état final » car, ici aussi, nous en simplifions la présentation. Tandis que le modèle analytique écrit en `Sophrosyne` peut amener à une infinité d'évolutions, le traitement effectué par l'interface graphique n'en conserve ici que la plus probable, ou la plus pertinente. La diversité des exécutions possibles d'une action peut toutefois être suggérée au moyen des éléments cartographiques associés : on opte alors ici pour un couloir plutôt qu'une ligne, là pour un ensemble d'éléments plutôt qu'un unique symbole.

Les éléments cartographiques et le calcul de l'état final d'une action ou mission dispose de leur propre petit langage de calcul. En guise d'illustration, nous présentons le calcul du couloir de vol lors d'une inspection linéaire, prenant comme paramètre pertinents pour le développement qui suit la latitude `lat` et la longitude `lon` de la destination, ainsi qu'une largeur de couloir `width` dans lequel l'inspection s'effectue convenablement. La situation est décrite par la figure 9.11, et le calcul du couloir de vol comme élément cartographique associé à la mission d'inspection linéaire est donné ci-dessous.

```
{
  "name": "Couloir de vol",
  "category": "geofence.in",
  "geometry": {
    "type": "Polygon",
    "coordinates": {
      "transform": "buffer",
```

```
{
    "name": "Nom du système",
    "variables": [
        {
            "name": "latitude",
            "type": "Number"
        },
        {
            "name": "longitude",
            "type": "Number"
        },
        {
            "name": "altitude",
            "type": "Number"
        }
    ],
    "actions": [
        {
            "type": "takeoff",
            "params": [
                {
                    "name": "alt",
                    "type": "Number"
                }
            ],
            "features": [
                {
                    "name": "Point de décollage",
                    "category": "waypoint.start",
                    "geometry": {
                        "type": "Point",
                        "coordinates": ["@longitude", "@latitude", "@altitude"]
                    }
                }
            ],
            "nextState": {
                "altitude": "$alt",
                "latitude": "@latitude",
                "longitude": "@longitude"
            }
        }
    ]
}
```

FIGURE 9.10 – Représentation d'un système `Sophrosyne` en `json`

```
      "values": [
        [["@lon", "@lat"],
         ["$lon", "$lat"]],
        "$width"
      ]}}
}
```

Les calculs font appel à deux types de variables :

- les variables d'état actuel (au commencement de l'action) du drone préfixées par `@` rappelant la notation `sophrosyne`, par exemple `"@lon"` pour la variable `lon`
- les paramètres de l'action ou de la mission préfixés par `$`, par exemple `"$width"` par le paramètre `width`

Ces variables peuvent être utilisées par diverses fonctions. Celles-ci sont appelées en passant un objet, ayant pour attribut *transform* le nom de la fonction, et pour attribut *values* la liste des arguments. Ainsi, nous pouvons écrire pour les opérateurs usuels

```
{"transform": "+", "values": [4, 6]}
```

ce qui, une fois interprété, renverra 10. En plus des opérateurs usuels, de nombreuses fonctions s'avèrent utiles pour nos applications. La transformation *buffer* par exemple prend deux arguments : les coordonnées d'une *LineString*, et une épaisseur. Elle renvoie un polygone circonscrivant cette ligne de part et d'autre jusqu'à l'épaisseur donnée. Dans la figure 9.11, le couloir de vol est ainsi le *buffer* d'une ligne coïncidant avec la trajectoire sur la figure et définie par ses extrémités à partir de la position initiale `["@lon", "@lat"]` et de la destination `["$lon", "$lat"]` ; la dernière dimension, c'est-à-dire la largeur du couloir, est donné par le dernier argument `"$width"`.

Nous ne détaillons pas ici les autres fonctions auxquelles nous recourons. Pour celles qui impliquent des calculs géographiques (parmi lesquelles *buffer*), nous nous appuyons sur l'extension `PostGIS` [16] de la base de donnée `PostgreSQL` fournissant des fonctionnalités de système d'information géographique particulièrement appréciables pour la manipulation de données hétérogènes telles que des coordonnées géographiques angulaires en WGS84 et des mètres.

```
{
    "name": "Nom de la mission",
    "initialState": {
        "altitude": 330,
        "latitude": 48.46864,
        "longitude": 6.10776
    },
    "mission": [
        {
            "id": 0,
            "type": "ACTION",
            "payload": {
                "type": "takeoff",
                "params": [{"name": "alt", "value": 10}]
            }
        }
    ]

}
```

FIGURE 9.12 – Représentation d'une mission `Sophrosyne` en `json`

FIGURE 9.11 – Capture d'écran de la planification d'une inspection de clôture aéroportuaire.

TABLE 9.2 – Correspondance des couleurs des éléments cartographiques de la figure 9.11

Couleur	**descriptif**
rose	clôture
bleu	trajectoire prévue
vert	couloir de vol (*geofence*)

Contrairement au système `json` et sa contrepartie en `sophrosyne`, la mission `json` reste proche de la mission `sophrosyne`, en particulier de son arbre syntaxique abstrait du fait de la présentation sous forme d'objet. En l'état actuel, seule la composition séquentielle d'actions et de missions est possible. La limitation apparente de l'expressivité peut toutefois être contournée par l'écriture de bibliothèques dissimulant les structures manquantes nécessaires dans des missions. Cette astuce suppose que l'on connaissent aux préalables tous les contextes et autres affectations ou abstractions qui seront utiles à l'écriture des missions. Cela nous semble être un faible désagrément devant l'avantage qui en procède : nous pouvons ainsi présenter une interface plus simple et plus concise au planificateur comme évoqué précédemment (cf section 9.2.2).

À partir des actions et missions définie par le système `json`, et d'une mission `json` les instanciant et les composant séquentiellement, nous produisons un calque décrivant la mission attendue : pour chaque action ou mission, les éléments cartographiques associés sont produits, et la valeur des variables d'état du système est actualisée selon l'attribut `"nextState"`. Nous l'avons mentionné précédemment (section 9.3.1), ce processus est une simulation MIL simpliste. Cette simulation est appréciable pour détecter des erreurs grossières, à l'instar du survol de zones interdites ; toutefois, l'obtention de garanties plus fortes se fera par des simulations plus réalistes ou par la preuve formelle de la correction de la mission. Le calque de mission appuie également la gestion des risques : la proximité d'un chemin rural indique un risque à la personne. Enfin, pour une estimation de la qualité de la modélisation, nous pouvons comparer les informations que nous donne cette simulation avec des traces d'exécutions réelles. Nous montrons cela dans la figure 9.13. Ces données ont été obtenues dans le cadre d'une expérimentation `ceos` d'inspection de conduites forcées à Saint-Dalmas. La mission prévoyait un vol à 1 m/s, et la trace d'exécution présentée est échantillonnée à 1 seconde. Dû au caractère expérimental de la plateforme, le télé-pilote amenait le drone jusqu'au premier point après le grillage. Le télé-pilote changeait alors le mode de l'autopilote, déclenchant la prise en main par le module de navigation `Sophrosyne` (voir figure C.2). La mission correspond alors à l'inspection sur quelques dizaines de mètres de la conduite forcée, avant de revenir au point de décollage en repassant par le premier point de passage.

FIGURE 9.13 – Superposition du calque de mission et de la trace d'exécution du système réel lors d'une inspection de conduites forcées. La largeur du couloir de vol (en vert) est d'un mètre de part et d'autre de la trajectoire attendue (en bleu). La trace d'exécution (en fuchsia) est difficilement discernable tant elle coïncide avec la trajectoire attendue.

9.4 Perspectives

L'application de planification de mission que nous présentons est une preuve de concept de l'intégration de `Sophrosyne` dans une chaîne logicielle outillée visant un public plus large. Le développement d'une telle application est par bien des aspects un travail prométhéen tant sa pertinence et son accessibilité sont étroitement liées à des fonctionnalités annexes, telles que la gestion de projet, ou la gestion des systèmes géodésiques. La liste des fonctionnalités que nous considérons ne sera pas détaillée ici du fait de sa longueur. C'est aussi à ce titre que nous avons renoncé à l'exhaustivité sur celles déjà implémentées pour nous concentrer sur celles touchant plus directement à la définition de mission avec `Sophrosyne`.

`Sophrosyne` comprend deux représentations graphiques de ses missions : les partitions et les calques de missions. Ces deux rendus, malgré leurs similitudes, sont des processus disjoints. Dans un premier temps, il s'agirait de vérifier la cohérence de ces deux représentations, en commençant par l'état final des actions. De plus, nous avons évoqué au chapitre 6 notre volonté de reprendre le processus de raisonnement sur le modèle analytique avec un langage dédié, servant à définir des stratégies pour le solveur, à en insérer les résultats pertinents dans la preuve formelle, et à l'établissement des partitions de missions. Le langage pourrait avoir comme produit secondaire la spécification des modèles d'éléments cartographiques associés à chaque action.

Par ailleurs, les modèles analytiques offrent encore un large potentiel inexploité. Nous avons justifié la séparation que nous opérons entre le modèle analytique du système et la bibliothèque d'exécution `pysophrosyne` (voir paragraphe 6.2.1). Mais si nous n'observons pas l'adéquation entre le modèle et l'exécution en embarqué, il serait pertinent d'effectuer cette analyse sur les traces d'exécution dont le traitement reste en l'état rudimentaire. Ceci pourrait se faire de manière statique dans un premier temps (au chargement des journaux d'exécution), puis dynamiquement si nous rajoutons un serveur de communication avec le module embarqué permettant de suivre l'évolution du drone sur l'application, la promouvant ainsi en une véritable station de contrôle au sol.

Conclusion et perspectives

Le passage de la robotique industrielle traditionnelle au système cyber-physique n'est pas anodin, et nécessite de repenser des méthodes et des outils pour leur programmation. Parmi toutes les problématiques qui émergent, nous nous sommes concentrés dans cet ouvrage sur deux axes qui procèdent de la volonté de ne pas avoir d'humain supervisant chaque système et de permettre à ces systèmes d'évoluer dans des environnements non maîtrisés :

- l'autonomisation, par la programmation du système cyber-physique
- la gestion des risques, en associant à la mission des mécanismes de supervision, de mitigation, et des méthodes d'analyse statique des missions

Ces travaux ne sauraient se justifier sans leur donner corps, tout d'abord par leur implémentation au sein d'outils dédiés, puis par la mise en oeuvre de ces outils pour la programmation de systèmes réels. Dans notre cas, le domaine d'application retenu est la programmation de drone, et il se concrétise notamment par l'utilisation des solutions proposées pour l'inspection d'ouvrages d'opérateurs d'importance vitale dans le projet céos. Notre objectif d'application de nos méthodes s'est traduit en la volonté de proposer une chaîne logicielle complète pour la programmation des systèmes cyber-physiques mobiles comprenant :

1. la définition de mission
2. l'analyse formelle de mission
3. l'exécution de mission
4. la simulation de mission
5. l'analyse des traces d'exécution

Ce chapitre conclusif revient sur nos contributions et résume les perspectives de notre travail.

Contributions

La démarche que nous proposons pour traiter les deux thématiques que nous avons identifiées (autonomisation et gestion des risques) est de combiner ces deux axes au sein d'une réponse unique. Celle-ci s'articule autour d'un langage de programmation dédié, `Sophrosyne` (cf chapitre 4). Nous n'avons certes pas ambitionné d'atteindre l'autonomie dans son acception la plus noble, qui suppose une capacité de réaction à des situations imprévues par une optimisation téléologique. Cependant, entre cette vision et la simplicité des principales solutions dédiées à la programmation de drone, qui ne permettent pas de prévoir des comportements en fonction de l'environnement ou de l'exécution de la mission, une automatisation de vols complexes reste à développer et notre travail s'inscrit dans ce cadre.

L'autonomisation peut être perçue comme le déploiement de diverses branches d'exécution associées à un mécanisme de sélection de la branche à exécuter. `Sophrosyne` utilise à l'effet de cette sélection l'évaluation de formules propositionnelles définies lors de la programmation de la

mission. Si ces propositions peuvent servir la finalité de la mission (*e.g.* répéter une tâche jusqu'à satisfaction d'une condition), elles intègrent également la problématique de gestion des risques en segmentant l'espace des états du système (*e.g.* le drone est-il dans son couloir de vol autorisé ?) et en y associant des comportements de mitigation du risque (*e.g.* s'il n'est pas dans son couloir, il doit y retourner). Ainsi, le premier trait caractéristique de `Sophrosyne` est sa triple structure de contrôle de flux d'exécution que nous avons nommée contexte.

La seconde singularité du langage est son traitement du temps. Le temps physique des horloges est une construction commode car précise et communicable, mais son utilisation comme base universelle est contraignante et tend éventuellement à complexifier la modélisation des systèmes. Ceci s'observe notamment lorsque des approximations sont introduites dans le modèle, et il est souvent plus efficace d'écrire qu'une action prend entre 9 et 11 secondes à être effectuée plutôt que de calculer les encadrements horaires des variables qui résultent. De ce constat nous introduisons une variable décrivant l'évolution, par rapport à laquelle la dérivation des variables d'état est définie.

L'écriture de programmes `Sophrosyne` comprend deux étapes. La première consiste en la modélisation du système considéré, et implique la description de l'état du système et les possibilités d'évolution de cet état selon l'action entreprise. La seconde est la programmation, en combinant ces actions au sein de programmes de mission, notamment à l'aide des contextes. Le modèle analytique du système crée la possibilité d'analyser les missions. Pour cela, nous avons tout d'abord explicité la sémantique formelle du langage (chapitre 5). De là, deux outils d'analyse ont été proposés :

- la partition de mission, offrant une représentation graphique des états théoriquement accessibles par l'exécution de la mission (paragraphe 6.3)
- la preuve d'une mission, reposant sur la notion de correction d'une mission que nous avons introduite à cet effet (chapitre 8)

Ces deux outils nécessitent de connaître l'état à tout instant de la mission. L'accès à cette information se fait par la résolution des systèmes différentiels associés à chaque action. L'état de l'art des systèmes de calcul formel ne permet pas de résoudre en toute généralité ces systèmes, et l'heuristique reste à cet égard la principale fonction à leur faire défaut. Pour cela, nous avons mené des travaux prospectifs visant à développer un solveur de systèmes différentiels construit sur la bibliothèque `Python` de calcul symbolique `sympy`. Cette première implémentation est encourageante, nous avons par exemple démontré sa pertinence sur un exemple de mission d'inspection comprenant des équations différentielles polynomiales, trigonométriques, et des encadrements de variables. Par ailleurs, la correction d'une mission comprend deux volets : la vivacité et la sûreté. La vivacité se décompose elle-même en une vivacité atomique, s'assurant qu'une évolution est toujours possible (*e.g.* la batterie du drone ne se décharge pas entièrement), et une vivacité séquentielle, donnant des garanties sur la transition entre action (*e.g.* le drone ne reçoit pas de consigne d'atterrissage s'il n'est pas en vol). La sûreté vérifie qu'il n'existe pas d'exécution qui sciemment invalide une propriété critique, à l'instar d'une *geofence* ou d'une décharge de la batterie sous un seuil critique. Nous avons donné un moyen d'évaluer la correction d'une mission en explicitant la dérivation des obligations de preuve afférentes exprimées en logique dynamique différentielle.

Les missions `Sophrosyne` reposent sur l'utilisation des contextes, des structures de contrôle de flux peu usuelles. Le passage d'un code `Sophrosyne` à un programme exécutable n'est donc pas une tâche triviale, ce qui a motivé l'introduction d'une compilation de programme `Sophrosyne` en un code exécutable `Python`. Une bibliothèque d'exécution a été écrite, `pysophrosyne`, permettant de maintenir la lisibilité du code généré (paragraphe 6.2). La transformation d'un code `Sophrosyne` en un programme exécutable `Python` est intégralement automatique pour une mis-

sion, et produit un *template* spécifiant les interfaces pour un système. Des programmes ainsi générés ont été exécutés en embarqué sur des drones, respectant notamment des exigences non-fonctionnelles réalistes qui avaient été définies pour les opérations concernées.

Ces différents éléments constituent la colonne vertébrale des méthodes et outils que nous proposons pour la programmation des systèmes cyber-physiques. La pertinence de tels outils dépend également de leur potentiel d'intégration au sein de solutions facilitant et étendant leur usage. Nous avons pour cela développé une application permettant à un profane en programmation de définir une mission et de partiellement l'analyser (chapitre 9). Pour cela, il peut s'avérer nécessaire de lui fournir préalablement un modèle du système et une bibliothèque définissant les comportements complexes. L'analyse proposée repose sur une version simplifiée du calcul des états accessibles projetés sur un calque de Système d'Information Géographique, et sur l'exécution de la mission dans un environnement de (co-)simulation avec logiciel ou matériel dans la boucle. L'ambition de cette application est de fournir une interface homme-machine unique pour définir, vérifier, tester, visualiser, déployer, et analyser les traces d'exécutions d'une mission `Sophrosyne`. Ces étapes ont été réalisées à divers niveau de complétion, ce que résume le tableau 3.

Table 3 – Avancement des principales fonctions de l'application de planification de mission

étape	**travail effectué**	**exemples d'améliorations envisagées**
Définir	Interface de définition de mission	Extension à la définition de contextes
		Programmation du système
Vérifier	Calque de mission	Assistant de preuve pour la correction de mission
		Partition de mission
Tester	Simulation SITL de mission PX4	Rétroaction de la co-simulation avec le traitement d'image de la visualisation
Visualiser	SIG avec calque de mission et données pertinentes	Génération dynamique de monde avec relief (simulation)
	Rendu graphique de la simulation	
Déployer	Génération de code exécutable de mission	Génération de code exécutable de système
Analyser les traces	Calque des journaux d'exécution	Confrontation des journaux à la partition de mission
		Diversification des éléments cartographiques

L'avancement des différentes briques suffit à démontrer le potentiel de la solution proposée. Celle-ci a été utilisée avec succès sur le projet céos pour assurer la planification et l'exécution de la navigation, ainsi que la production d'éléments du rapport d'inspection au moyen des journaux d'exécution. La vérification formelle de mission a toutefois été écartée sur ce projet, du fait des limitations des logiciels tiers employés. Pour autant, la notion de correction de mission est définie et nous avons illustré sa mise en oeuvre sur des exemples analogues. Nous avons ainsi proposé une méthode de programmation de systèmes cyber-physiques, construite autour d'un langage dédié. Cette méthode accroît l'autonomie des systèmes considérés, en permettant la définition de comportements complexes adoptés selon la situation. Ce déclenchement peut notamment intervenir dans un cadre de gestion des risques, le contrôle de flux d'exécution s'effectuant par des

structures de supervision. Autour de cette méthode nous avons construit des solutions logicielles. Ces outils comprennent non seulement une implémentation des divers constituant de la méthode que nous proposons, mais également des extensions améliorant son accessibilité et son utilisabilité.

Perspectives

Le développement d'une telle solution logicielle est un travail de Sisyphe : chaque application met en lumière de nouvelles fonctionnalités qui seraient souhaitables pour améliorer l'expérience utilisateur. En plus de devoir abandonner tout espoir d'exhaustivité, établir une telle liste n'aurait ici que peu d'intérêt. Nous nous contenterons des quelques pistes d'améliorations envisagées présentées dans le tableau 3. Celles-ci ne concernent généralement pas directement `Sophrosyne`, à l'instar de la génération dynamique d'un monde avec relief qui est un défi pour le rendu physique et graphique, ou l'extension à la définition de contextes qui n'est qu'une question d'implémentation si les exigences de représentation restent faibles.

De fait, ce qui constitue la principale limite de notre contribution (c'est-à-dire son plus grand défi) est la vérification de mission. Le chapitre 8 a montré un complément de la méthode par l'ajout de perturbations. En plus de permettre d'analyser des pans de mission autrement inaccessibles, les perturbations initient une quantification de la résilience de la mission. Par ailleurs, bien que nous en ayons établi les fondations, la preuve formelle de mission reste fastidieuse ou incomplète du fait des inadéquations restantes entre les moyens utilisés pour démontrer leur faisabilité et les objectifs ambitionnés. Nous allons discuter de trois de ces limites :

- la prolifération des obligations de preuves
- la complexité des modèles
- la spécialisation pour `Sophrosyne` de la logique dynamique différentielle

Chaque nouvelle action d'une mission entraîne au minimum deux nouvelles preuves à effectuer (vivacité atomique et séquentielle). Si cette action intervient dans la mission principale d'un contexte, une preuve de sûreté peut de plus être demandée. À ces preuves s'ajoutent encore celles évoquées de perturbation de l'exécution, qui multiplient les obligations de preuve nécessaires. Cette prolifération est actuellement contraignante pour vérifier des missions même relativement simples. Par ailleurs, les modèles à prouver croissent linéairement en le nombre d'action, rendant la tâche d'autant plus fastidieuse. Il est toutefois possible de s'armer en conséquence pour rendre ces deux aspects inoffensifs. Nous avons défini la dérivation des obligations de preuve par induction, et il apparaît clairement que tous ces modèles partagent des préfixes. Chaque preuve effectuée doit donc pouvoir servir comme lemme pour simplifier la preuve suivante, ne laissant que peu de travail sur le séquent résultant. Une deuxième piste de simplification et l'utilisation des langages de tactique. Le traitement effectué sur chaque action est généralement le même : il s'agit d'expliciter l'état du système en fonction du paramètre γ afin d'obtenir des invariants de boucle. De là, en extrayant la tactique ayant servi à prouver les séquents afférents à une action, de nombreux séquents devraient se résoudre simplement par l'application de cette tactique. La troisième piste concerne l'utilisation du solveur que nous avons développé pour raffiner des modèles. Pour des modèles que nous ne pouvons pas exprimer directement en logique dynamique différentielle (*e.g.* fonctions trigonométriques), l'objectif est alors de produire des modèles équivalents (*e.g.* formulation différentielle polynomiale correspondante) ou généralisants. Ces utilisations présupposent d'évaluer minutieusement sa correction selon les cas d'application, mais, à défaut, le résultat du calcul de l'état du système par le solveur est candidat comme invariant de boucle (il reste alors à prouver cette invariance).

Enfin, nous avons évoqués au chapitre 8 le projet de développer un assistant de preuve dédié

aux modèles en logique dynamique différentielles de missions `Sophrosyne`. Nous ajoutons ici une autre motivation que celles déjà énoncées : les modèles analytiques des actions `Sophrosyne` confèrent à γ un statut particulier du fait de son évolution ($\gamma' = 1 \land 0 \leq \gamma \leq 1$). Le calcul de l'évolution des variables d'état en fonction de γ est pour nous un objectif, ce que ne perçoit pas un assistant de preuve généraliste comme `KeYmaera X` et son *backend* (`Z3` ou `wolfram`). Pour ces derniers, γ est traité comme une variable au même titre que l'altitude du système, ce qui affecte leur capacité de résolution des équations différentielles. La conception d'un assistant de preuve dédié mettant à profit une version étendue de notre solveur comme *backend* permettrait de faciliter le processus de preuve et d'élargir son champ d'application.

Le renforcement de la partie formelle est ainsi l'axe de développement de `Sophrosyne` que nous privilégions. À celui-ci s'ajoutent des évolutions plus spécifiques qui apparaîtront par l'usage. Parmi ces dernières, la programmation de systèmes cyber-physiques mis en réseau et concourrant à une même mission doit encore être éprouvée. `Sophrosyne` dispose d'ores et déjà de fonctionnalités permettant à diverses entités de communiquer entre elles : pour l'essentiel, des entrées provenant d'un capteur de communication ou d'un système de géo-localisation peuvent traitées de manière analogue, au moyen de variables d'état. Le passage d'une exécution de mission par un unique système à celle par un réseau ne devrait donc pas nécessiter de grande adaptation. En revanche, la vérification de mission pourrait être impossible en considérant ces missions séparément, et `Sophrosyne` devrait prévoir un mécanisme de preuves concurrentes. Cela n'a pas que des implications bénignes : nous avons opté pour γ comme variable principale, et synchroniser les horloges des différents modèles requiert *a priori* d'inverser les systèmes. L'utilisation de `Sophrosyne` pour programmer un réseau de systèmes cyber-physiques entraînera donc non seulement des évolutions des outils, mais également de la méthode pour la vérification formelle.

Annexes

Annexe A

La grammaire de Sophrosyne

Nous donnons ci-dessous la grammaire sous forme de Backus-Naur de `Sophrosyne`. Par convention, les éléments écrits en majuscule correspondent à des terminaux. Ces éléments sont transparents et nous ne les rappelons pas ; précisons toutefois que $\langle PARAM \rangle$ correspond au paramètre γ, remplacé par '_' lors de l'écriture d'un programme. Les mots-clés, opérateurs et délimiteurs du langage sont écrits en gras ou entre guillemets.

expression

$\langle expr \rangle$: := $\langle numexpr \rangle$ | $\langle formula \rangle$ | $\langle record \rangle$

$\langle args \rangle$: := $\langle ID \rangle$ | $\langle args \rangle$ **,** $\langle ID \rangle$

$\langle field \rangle$: := $\langle ID \rangle$ **:** $\langle formula \rangle$ | $\langle ID \rangle$ **:** $\langle numExpr \rangle$ | $\langle ID \rangle$ **:** $\langle record \rangle$

$\langle fieldList \rangle$: := $\langle fieldList \rangle$ **,** $\langle field \rangle$ | $\langle field \rangle$

$\langle record \rangle$: := **{** $\langle fieldList \rangle$ **}**

$\langle exprList \rangle$: := $\langle expr \rangle$ **,** $\langle exprList \rangle$ | $\langle expr \rangle$ **,** $\langle expr \rangle$ | $\langle expr \rangle$ **,**

$\langle formula \rangle$: := $\langle boolean \rangle$ | $\langle name \rangle$ | $\langle dottedId \rangle$ | $\langle application \rangle$
| $\langle STATEVAR \rangle$ | $\langle dottedStatevar \rangle$
| **not** $\langle formula \rangle$ | $\langle numExpr \rangle$ < $\langle numExpr \rangle$ | $\langle numExpr \rangle$ > $\langle numExpr \rangle$
| $\langle numExpr \rangle$ <= $\langle numExpr \rangle$ | $\langle numExpr \rangle$ >= $\langle numExpr \rangle$ | $\langle numExpr \rangle$ = $\langle numExpr \rangle$
| $\langle numExpr \rangle$ <> $\langle numExpr \rangle$ | $\langle formula \rangle$ **or** $\langle formula \rangle$ | $\langle formula \rangle$ **and** $\langle formula \rangle$
| **(** $\langle formula \rangle$ **)**

$\langle numExpr \rangle$: := $\langle number \rangle$ | $\langle name \rangle$ | $\langle dottedId \rangle$ | $\langle application \rangle$
| $\langle STATEVAR \rangle$ | $\langle dottedStatevar \rangle$ | $\langle PARAM \rangle$
| **-** $\langle numExpr \rangle$ | $\langle numExpr \rangle$ + $\langle numExpr \rangle$ | $\langle numExpr \rangle$ **-** $\langle numExpr \rangle$
| $\langle numExpr \rangle$ ***** $\langle numExpr \rangle$ | $\langle numExpr \rangle$ / $\langle numExpr \rangle$ | $\langle numExpr \rangle$ ****** $\langle numExpr \rangle$

$\langle property \rangle$: := $\langle boolean \rangle$ | $\langle name \rangle$ | $\langle dottedId \rangle$ | $\langle application \rangle$
| $\langle STATEVAR \rangle$ | $\langle dottedStatevar \rangle$
| **not** $\langle property \rangle$ | $\langle extNumExpr \rangle$ < $\langle extNumExpr \rangle$ | $\langle extNumExpr \rangle$ > $\langle extNumExpr \rangle$
| $\langle extNumExpr \rangle$ <= $\langle extNumExpr \rangle$ | $\langle extNumExpr \rangle$ >= $\langle extNumExpr \rangle$ | $\langle extNumExpr \rangle$ = $\langle extNumExpr \rangle$
| $\langle extNumExpr \rangle$ <> $\langle extNumExpr \rangle$ | $\langle property \rangle$ **or** $\langle property \rangle$ | $\langle property \rangle$ **and** $\langle property \rangle$
| **(** $\langle property \rangle$ **)**

⟨*extNumExpr*⟩ : := ⟨*number*⟩ | ⟨*name*⟩ | ⟨*dottedId*⟩ | ⟨*application*⟩
| ⟨*STATEVAR*⟩ | ⟨*dottedStatevar*⟩ | ⟨*PARAM*⟩ | ⟨*derived*⟩
| **-** ⟨*extNumExpr*⟩ | ⟨*extNumExpr*⟩ + ⟨*extNumExpr*⟩ | ⟨*extNumExpr*⟩ **-** ⟨*extNumExpr*⟩
| ⟨*extNumExpr*⟩ ***** ⟨*extNumExpr*⟩ | ⟨*extNumExpr*⟩ / ⟨*extNumExpr*⟩ | ⟨*extNumExpr*⟩ ****** ⟨*extNumExpr*⟩
| **(** ⟨*extNumexpr*⟩ **)**

⟨*boolean*⟩ : := **True** | **False**

⟨*number*⟩ : := ⟨*INTNUMBER*⟩ | ⟨*FLOATNUMBER*⟩

⟨*derived*⟩ : := ⟨*STATEVAR*⟩ **'** | ⟨*dottedStatevar*⟩ **'**

⟨*application*⟩ : := ⟨*ID*⟩ **( )** | ⟨*ID*⟩ **(** ⟨*expr*⟩ **)** | ⟨*ID*⟩ **(** ⟨*exprList*⟩ **)**

⟨*dottedId*⟩ : := ⟨*name*⟩ **.** ⟨*ID*⟩ | ⟨*dottedId*⟩ **.** ⟨*ID*⟩

⟨*dottedStatevar*⟩ : := ⟨*STATEVAR*⟩ **.** ⟨*ID*⟩ | ⟨*dottedStatevar*⟩ **.** ⟨*ID*⟩

⟨*name*⟩ : := ⟨*ID*⟩

Instructions

⟨*statevarDecl*⟩ : := ⟨*STATEVAR*⟩ **:** ⟨*expr*⟩ | ⟨*statevarDecl*⟩ **,** ⟨*STATEVAR*⟩ **:** ⟨*expr*⟩

⟨*invariant*⟩ : := ⟨*property*⟩ | ⟨*invariant*⟩ **,** ⟨*property*⟩

⟨*requirement*⟩ : := **requires** ⟨*formula*⟩

⟨*actionBlock*⟩ : := ⟨*property*⟩ | ⟨*stmt*⟩ ⟨*actionBlock*⟩

⟨*actionDef*⟩ : := **action** ⟨*ID*⟩ **:** ⟨*requirement*⟩ ⟨*actionBlock*⟩ **;**
| **action** ⟨*ID*⟩ **(** args **) :** ⟨*requirement*⟩ ⟨*actionBlock*⟩ **;**

⟨*actionsDef*⟩ : := ⟨*empty*⟩ | ⟨*actionDef*⟩ | ⟨*actionsDef*⟩ ⟨*actionDef*⟩

⟨*assignment*⟩ : := ⟨*ID*⟩ ← ⟨*expr*⟩ | ⟨*dottedId*⟩ ← ⟨*expr*⟩

⟨*abstraction*⟩ : := ⟨*ID*⟩ → ⟨*expr*⟩ | ⟨*dottedId*⟩ → ⟨*expr*⟩

⟨*functionBlock*⟩ : := ⟨*stmt*⟩ | ⟨*functionBlock*⟩ ⟨*stmt*⟩
| ⟨*functionCxt*⟩ | ⟨*functionBlock*⟩ ⟨*functionCxt*⟩

⟨*functionCxt*⟩ : := **with** ⟨*formula*⟩ **:** ⟨*functionBlock*⟩ **else :** ⟨*functionBlock*⟩ **;**

⟨*functionDef*⟩ : := **def** ⟨*ID*⟩ **:** ⟨*functionBlock*⟩ **;**
| **def** ⟨*ID*⟩ **(** ⟨*args*⟩ **) :** ⟨*functionBlock*⟩ **;**

⟨*context*⟩ : := **with** ⟨*expr*⟩ **:** ⟨*mission*⟩ **else :** ⟨*mission*⟩ **;**

⟨*labelling*⟩ : := **label** ⟨*ID*⟩

⟨*reference*⟩ : := **resume** ⟨*ID*⟩ | **resume**

⟨*missionStmt*⟩ : := ⟨*context*⟩ | ⟨*stmt*⟩ | ⟨*functionDef*⟩ | ⟨*application*⟩ | ⟨*missionDef*⟩ |
| ⟨*labelling*⟩ | ⟨*reference*⟩

⟨*mission*⟩ : := ⟨*missionStmt*⟩ | ⟨*mission*⟩ ⟨*missionStmt*⟩

⟨*missionDef*⟩ : := **mission** ⟨*ID*⟩ **:** ⟨*mission*⟩ **;**
| **mission** ⟨*ID*⟩ **(** args **) :** ⟨*mission*⟩ **;**

⟨*systemDef*⟩ : := **system** ⟨*ID*⟩ **: {** ⟨*statevarDecl*⟩ **} invariant : {** ⟨*invariant*⟩ **}** ⟨*actionsDef*⟩ **;**

⟨*systemDecl*⟩ : := **system** ⟨*ID*⟩

⟨*import*⟩ : := **import** ⟨*ID*⟩ | **import** ⟨*dottedId*⟩

⟨*importList*⟩ : := ⟨*empty*⟩ | ⟨*importList*⟩ ⟨*import*⟩

⟨*program*⟩ : := ⟨*importList*⟩ ⟨*systemDef*⟩ | ⟨*importList*⟩ ⟨*systemDecl*⟩ ⟨*mission*⟩

Annexe B

Les règles de la logique dynamique différentielle

Les différentes règles du calcul des séquents de la logique dynamique différentielle sont listées ci-dessous. Elles sont reprises de [151], et nous réemployons des notations introduites au chapitre 7. `skolem` est une fonction de skolemnisation, instanciant des variables quantifiées. `QE` est une fonction d'élimination de quantificateurs.

$$\frac{\phi \vdash}{\vdash \neg\phi} \qquad \frac{\vdash \phi}{\neg\phi \vdash} \qquad \frac{\vdash \phi, \psi}{\vdash \phi \vee \psi} \qquad \frac{\phi \vdash \qquad \psi \vdash}{\phi \vee \psi \vdash} \qquad \frac{\vdash \phi \qquad \vdash \psi}{\vdash \phi \wedge \psi}$$

$$\frac{\phi, \psi \vdash}{\phi \wedge \psi \vdash} \qquad \frac{\phi \vdash \psi}{\vdash \phi \to \psi} \qquad \frac{\vdash \phi \qquad \psi \vdash}{\phi \to \psi \vdash} \qquad \frac{}{\phi \vdash \phi} \qquad \frac{\vdash \phi \qquad \phi \vdash}{\vdash}$$

$$\frac{\langle\alpha\rangle\langle\beta\rangle\phi}{\langle\alpha;\,\beta\rangle\phi} \qquad \frac{\phi \vee \langle\alpha\rangle\langle\alpha^*\rangle\phi}{\langle\alpha^*\rangle\phi} \qquad \frac{\phi^{\theta_1 \ldots \theta_n}_{x_1 \ldots x_n}}{\langle x_1 := \theta_1 \ldots x_n := \theta_n\rangle\phi}$$

$$\frac{[\alpha][\beta]\phi}{[\alpha;\,\beta]\phi} \qquad \frac{\phi \wedge [\alpha][\alpha^*]\phi}{[\alpha^*]\phi} \qquad \frac{\langle x_1 := \theta_1 \ldots x_n := \theta_n\rangle\phi}{[x_1 := \theta_1 \ldots x_n := \theta_n]\phi}$$

$$\frac{\langle\alpha\rangle\phi \vee \langle\beta\rangle\phi}{\langle\alpha \cup \beta\rangle\phi} \qquad \frac{\chi \wedge \psi}{\langle ?\chi\rangle\psi} \qquad \frac{\exists t \geq 0\,((\forall 0 \leq s \leq t \langle S_s\rangle\chi) \wedge \langle S_t\rangle\phi)}{\langle x_1' = \theta_1, \ldots x_n' = \theta_n \wedge \chi\rangle\phi}$$

$$\frac{[\alpha]\phi \wedge [\beta]\phi}{[\alpha \cup \beta]\phi} \qquad \frac{\chi \to \psi}{[?\chi]\psi} \qquad \frac{\exists t \geq 0\,((\forall 0 \leq s \leq t \langle S_s\rangle\chi) \to \langle S_t\rangle\phi)}{[x_1' = \theta_1, \ldots x_n' = \theta_n \wedge \chi]\phi}$$

$$\frac{\vdash \phi(\mathrm{skolem}(X_1, \ldots, X_n))}{\vdash \forall x \phi(x)} \qquad \frac{\phi(\mathrm{skolem}(X_1, \ldots, X_n)) \vdash}{\exists x \phi(x) \vdash} \qquad \frac{\vdash \phi(X)}{\vdash \exists x \phi(x)} \qquad \frac{\phi(X) \vdash}{\forall x \phi(x) \vdash}$$

$$\frac{\vdash QE(\forall X(\phi(X) \vdash \psi(X)))}{\phi(\mathrm{skolem}(X_1, \ldots, X_n)) \vdash \psi(\mathrm{skolem}(X_1, \ldots, X_n))} \qquad \frac{\vdash QE(\exists X \bigwedge_i(\phi_i \vdash \psi_i))}{\phi_1 \vdash \psi_1 \ldots, \phi_n \vdash \psi_n}$$

$$\frac{\vdash \forall^\alpha(\phi \to \psi)}{[\alpha]\phi \vdash [\alpha]\psi} \qquad \frac{\vdash \forall^\alpha(\phi \to \psi)}{\langle\alpha\rangle\phi \vdash \langle\alpha\rangle\psi}$$

$$\frac{\vdash \forall^\alpha(\phi \to [\alpha]\phi)}{\phi \vdash [\alpha^*]\phi} \qquad \frac{\vdash \forall^\alpha \forall v > 0(\phi(v) \to \langle\alpha\rangle\phi(v-1)}{\exists v \phi(v) \vdash \langle\alpha^*\rangle\exists v \leq 0 \phi(v)}$$

Annexe C

Éléments de la navigation automatique `ceos`

Dans cette section nous présentons quelques éléments techniques de la navigation automatique développée pour le projet `ceos`.

C.1 Les fonctionnalités

La brique de navigation automatique assure les déplacements du drone lors d'une mission d'inspection `ceos`. Elle est effective entre le décollage et l'atterrissage exclus dans le cadre d'une exécution nominale. Ces inspections répondent à trois scénarios :

- inspection de ligne HTA
- inspection de conduite forcée
- inspection de clôture aéroportuaire

Les principales caractéristiques de la navigation automatique sont

- l'envoi de consignes de cinématique conforme à la mission donnée
- l'intégration d'un système de *geofencing* permettant de restreindre son enveloppe de vol

Un dispositif d'ajustement de trajectoire permet de palier des approximations du plan de vol (géo-référencement des objets d'intérêt) et de la géo-localisation du drone.

C.2 Les interfaces avec le système

L'architecture contextuelle de la navigation automatique est présentée dans la figure C.1. Avant exécution, la mission est intégralement générée par le système de planification de la navigation. Pendant l'exécution de la mission, la navigation interagit avec

- l'autopilote, pour obtenir l'état du drone et communiquer la cinématique désirée
- le module de surveillance et diagnostic, pour surveiller l'état de la navigation et informer du mode d'inspection
- le module d'asservissement de la nacelle, pour adapter la trajectoire et informer du mode d'inspection

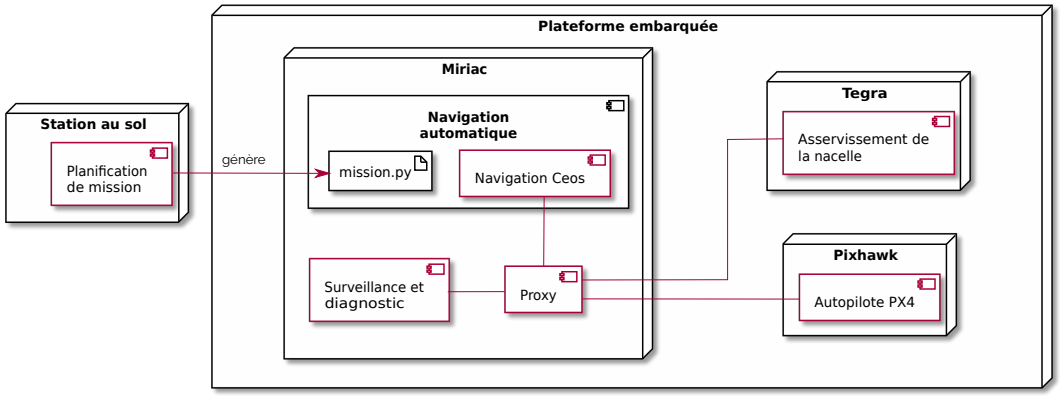


FIGURE C.1 – Architecture contextuelle de la navigation automatique

C.3 Les cas d'usages

La brique de navigation est prévue pour une utilisation dans le cadres des inspections du projet `ceos`. Sa mise en oeuvre générale est indépendante du cas d'usage, elle est présentée dans la figure C.2. Trois phases distinctes se succèdent

1. **l'initialisation**, qui est à la charge du télé-pilote. Cette phase, qui comprend la manoeuvre de décollage, se termine lorsque le télé-pilote enclenche le mode `offboard` de l'autopilote.
2. **l'inspection automatique**, durant laquelle la navigation automatique contrôle l'autopilote. Cette phase se termine par la complétion de la mission ou par l'interruption de la navigation par le télé-pilote.
3. **le contrôle manuel**, durant lequel l'opération se terminera, et qui comprend la manoeuvre d'atterrissage.

C.4 Un exemple d'exigence spécifique fonctionnelle

Trajectoire du drone – inspection linéaire

La navigation automatique doit fournir une trajectoire pour l'inspection d'objet d'intérêt linéaires (ligne HTA, conduite forcée, clôture aéroportuaire). La trajectoire est définie en vitesse le long du chemin, puis en position à proximité de la destination.

- *entrées* : le départ et l'arrivée du chemin (en coordonnées NED), la vitesse linéaire d'inspection
- *origine* : les entrées sont définies dans le script de mission généré par le système de planification
- *sorties* : une boucle de commande `MAVLink` de vitesse et lacet, puis position et lacet, désirés
- *destination* : autopilote

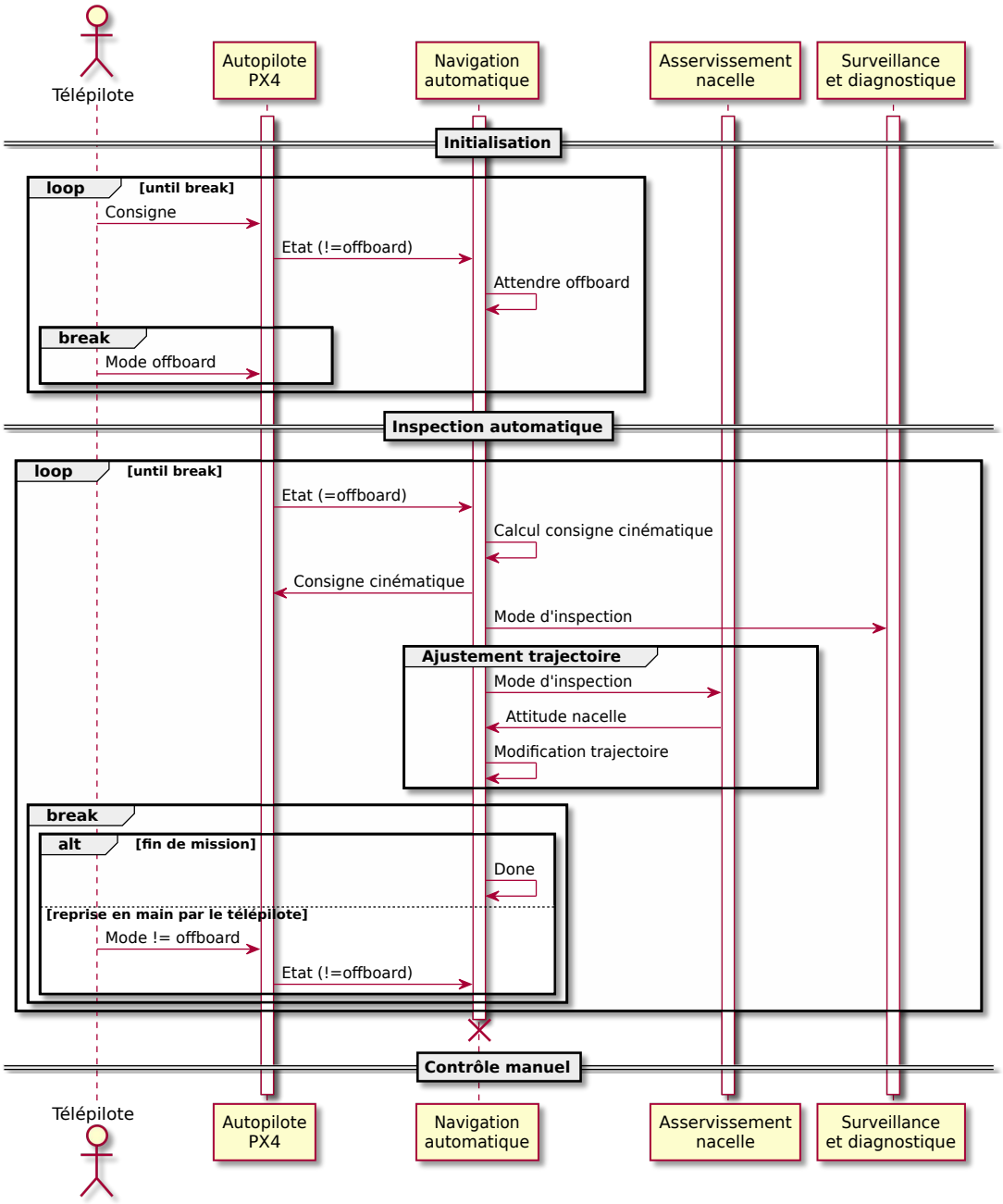


FIGURE C.2 – Diagramme de séquence des interactions de la navigation automatique

Bibliographie

[1] Allemagne : un drone perturbe un meeting d'angela merkel. *Le Figaro*. Consulté le : 16/11/2020.

[2] Amazon prime air. https://www.amazon.com/Amazon-Prime-Air/b?ie=UTF8&node=8037720011. Consulté le : 31/08/2020.

[3] Api openweathermap. https://openweathermap.org/api. Consulté le : 14/01/2021.

[4] Ardupilot. https://ardupilot.org/. Consulté le : 13/11/2020.

[5] Coppeliasim. https://www.coppeliarobotics.com/. Consulté le : 13/11/2020.

[6] Dronedefender de battelle. https://www.battelle.org/government-offerings/national-security/payloads-platforms-controls/counter-UAS-technologies/dronedefender. Consulté le : 16/11/2020.

[7] Dronekit. https://dronekit.io/. Consulté le : 13/11/2020.

[8] Drones : A vision has become reality. https://www.post.ch/en/about-us/innovation/innovations-in-development/drones. Consulté le : 16/11/2020.

[9] Gazebo. http://gazebosim.org/. Consulté le : 13/11/2020.

[10] Keymaera x theorem prover for hybrid systems. https://github.com/LS-Lab/KeYmaeraX-release.

[11] Lilium. https://lilium.com/. Consulté le : 16/11/2020.

[12] Mavlink. https://mavlink.io/en/. Consulté le : 13/11/2020.

[13] Mavsdk. https://mavsdk.mavlink.io/develop/en/index.html. Consulté le : 13/11/2020.

[14] Open policy agent. https://www.openpolicyagent.org/. Consulté le : 14/01/2021.

[15] Pixhawk. https://pixhawk.org/. Consulté le : 14/01/2021.

[16] Postgis. https://postgis.net/.

[17] Project wing. https://x.company/projects/wing/. Consulté le : 31/08/2020.

[18] Projet ceos. https://www.ceos-systems.com/fr/Projet-CEOS-Pour-L-Inspection-D-Ouvrages-Par-Drones.html. Consulté le : 14/01/2021.

[19] Python syntax and semantics. https://en.wikipedia.org/wiki/Python_syntax_and_semantics. Consulté le : 14/09/2020.

[20] Ros. https://www.ros.org/. Consulté le : 13/11/2020.

[21] Ros2. https://index.ros.org/doc/ros2/. Consulté le : 13/11/2020.

[22] Unreal engine. https://www.unrealengine.com/en-US/. Consulté le : 14/01/2021.

[23] Volocopter. https://www.volocopter.com/en/. Consulté le : 16/11/2020.

[24] Xd-motion. https://xd-motion.fr/reference-works/formula-1-grand-prix-de-france-2019/. Consulté le : 13/11/2020.

[25] Zipline. https://flyzipline.com/. Consulté le : 16/11/2020.

[26] Superman et un aéronef se crashent sur la centrale du bugey. https://www.greenpeace.fr/action-superman-survole-crash-centrale-nucleaire-bugey/, 03/07/2018. Consulté le : 16/11/2020.

[27] Israeli drones to patrol brazilian skies during world cup. https://www.haaretz.com/israel-news/business/israeli-drones-for-world-cup-1.5329127, 04/03/2014. Consulté le : 16/11/2020.

[28] Hôpitaux : livraison de médicaments par drones testée avec succès à lugano. *Le Nouvelliste* (04.10.2017). Consulté le : 16/11/2020.

[29] Repéré par un drone, un homme verbalisé après avoir promené son chat sur la plage sans attestation. *Ouest France* (07/04/2020). Consulté le : 16/11/2020.

[30] Airbus' skyways drone trials world's first shore-to-ship deliveries. https://www.airbus.com/newsroom/press-releases/en/2019/03/airbus-skyways-drone-trials-worlds-first-shoretoship-deliveries.html, 15/03/2019. Consulté le : 16/11/2020.

[31] L'armée de l'air enrôle des aigles royaux contre les drones. *Ouest France* (16/02/2017). Consulté le : 16/11/2020.

[32] 3DR. About 3dr. https://www.3dr.com/company/about-3dr. Consulté le 14/09/2020.

[33] ADOLF, F.-M., FAYMONVILLE, P., FINKBEINER, B., SCHIRMER, S., AND TORENS, C. Stream Runtime Monitoring on UAS. In *International Conference on Runtime Verification* (2018), no. March, pp. 33—-49.

[34] AFP. Israel's drone industry becomes global force. https://www.france24.com/en/20191128-israel-s-drone-industry-becomes-global-force, 28/11/2019. Consulté le : 13/11/2020.

[35] ALLIOT-MARIE, M. Intervention du Ministre de l'Intérieur, de l'Outre-Mer et des Collectivités Territoriales lors de l'inauguration de l'Hôtel de Police Lyon-Montluc. https://mobile.interieur.gouv.fr/Archives/Archives-ministres-de-l-Interieur/Archives-de-Michele-Alliot-Marie-2007-2009/Interventions/22.10.2007-Inauguration-de-l-Hotel-de-Police-Lyon-Montluc, 22/10/2007. Consulté le : 14/11/2020.

[36] ALMOHAMMAD, A., AND SPECKHARD, A. ISIS Drones : Evolution , Leadership , Bases , Operations and Logistics. *ICSVE Research Report* (2017).

[37] AMES, A. D., COOGAN, S., EGERSTEDT, M., NOTOMISTA, G., SREENATH, K., AND TABUADA, P. Control barrier functions : Theory and applications. In *18th European Control Conference* (2019), EUCA, pp. 3420–3431.

[38] ANDOVA, S., VAN DEN BRAND, M. G., ENGELEN, L. J., AND VERHOEFF, T. MDE basics with a DSL focus. *Lecture Notes in Computer Science 7320 LNCS* (2012), 21–57.

[39] ARDUPILOT DEV TEAM. Mission planner. https://ardupilot.org/planner/, 2020-09-15.

[40] AUBIN, J.-P., BAYEN, A. M., AND SAINT-PIERRE, P. *Viability Theory : New Directions*. No. August 2015. 2011.

[41] AUBIN, J.-P., AND CELLINA, A. *Differential inclusions*. 1984.

[42] Aubin, J. P., Lygeros, J., Quincampoix, M., Sastry, S., and Seube, N. Impulse differential inclusions : A viability approach to hybrid systems. *IEEE Transactions on Automatic Control 47*, 1 (2002), 2–20.

[43] Banach, R., Butler, M., Qin, S., Verma, N., and Zhu, H. *Core Hybrid Event-B I : Single Hybrid Event-B machines*, vol. 105. Elsevier B.V., 2015.

[44] Barthes, R. Mythologies, 1957.

[45] Barthes, R. Éléments de sémiologie. *Communications 4*, 1 (1964), 91–135.

[46] Bassi, E. European drones regulation : Today's legal challenges. *2019 International Conference on Unmanned Aircraft Systems, ICUAS 2019* (2019), 443–450.

[47] Beazley, D. Sly (sly lex yacc). `https://sly.readthedocs.io/en/latest/sly.html`.

[48] Belga. Nouvel An à Bruges : le spectacle de drones n'a pas pu être entièrement exécuté. `https://www.lesoir.be/270178/article/2020-01-01/nouvel-bruges-le-spectacle-de-drones-na-pas-pu-etre-entierement-execute`, 01/01/2020. Consulté le : 13/11/2020.

[49] Bemporad, A., and Morari, M. Control of systems integrating logic, dynamics, and constraints. *Automatica 35*, 3 (1999), 407–427.

[50] Bergounhoux, J. Ces 2016 : Le chinois ehang présente un drone géant pour transporter les gens. *L'Usine Digitale* (07/01/2016). Consulté le : 16/11/2020.

[51] Berry, G., and Gonthier, G. The Esterel synchronous programming language : design, semantics, implementation. *Science of Computer Programming 19* (1992), 87–152.

[52] Besada, J., Bergesio, L., Campaña, I., Vaquero-Melchor, D., López-Araquistain, J., Bernardos, A., and Casar, J. Drone Mission Definition and Implementation for Automated Infrastructure Inspection Using Airborne Sensors. *Sensors 18*, 4 (2018), 1170.

[53] Besada, J. A., Frontera, G., Crespo, J., Casado, E., and Lopez-Leones, J. Automated aircraft trajectory prediction based on formal intent-related language processing. *IEEE Transactions on Intelligent Transportation Systems 14*, 3 (2013), 1067–1082.

[54] Bézivin, J. In search of a basic principle for Model Driven Engineering. *Special Novatica Issue - UML and Model Engineering 5*, 2 (2004), 21–24.

[55] Bohrer, B., Tan, Y. K., Mitsch, S., Myreen, M. O., and Platzer, A. VeriPhy : Verified Controller Executables from Verified Cyber-Physical System Models. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation* (2018), pp. 617—-630.

[56] Bohrer, B., Tan, Y. K., Mitsch, S., Sogokon, A., and Platzer, A. A Formal Safety Net for Waypoint-Following in Ground Robots. *IEEE Robotics and Automation Letters 4*, 3 (2019), 2910–2917.

[57] Borelly, J.-J., Coste-Manière, E., Espiau, B., Kapellos, K., Pissard-Gibollet, R., Simon, D., and Nicolas, T. The Orccad Architecture. *The International Journal of Robotics Research 17*, 4 (1998), 338–359.

[58] Boukoberine, M. N., Zhou, Z., and Benbouzid, M. A critical review on unmanned aerial vehicles power supply and energy management : Solutions, strategies, and prospects. *Applied Energy 255*, September (2019), 113823.

[59] Bozhinoski, D., Di Ruscio, D., Malavolta, I., Pelliccione, P., and Tivoli, M. FLYAQ : Enabling non-expert users to specify and generate missions of autonomous multicopters. In *30th IEEE/ACM International Conference on Automated Software Engineering, ASE 2015* (2015), no. August, pp. 801–806.

[60] Brandl, G., and Pygments contributors. Pygments, python syntax highlighter. `https://pygments.org/`.

[61] Capitan, C., Capitan, J., Castano, A. R., and Ollero, A. Risk assessment based on SORA methodology for a UAS media production application. *2019 International Conference on Unmanned Aircraft Systems, ICUAS 2019*, March (2019), 451–459.

[62] Carreira, P., Amaral, V., and Vangheluwe, H. *Foundations of Multi-Paradigm Modelling for Cyber-Physical Systems*. Springer, 2020.

[63] Caseau, Y., Josset, F. X., and Laburthe, F. CLAIRE : Combining sets, search and rules to better express algorithms. *Theory and Practice of Logic Programming 2*, 6 (2002), 769–805.

[64] Champagnat, R., Esteban, P., Pingaud, H., and Valette, R. Petri net based modeling of hybrid systems. *Computers in Industry 36*, 1-2 (1998), 139–146.

[65] Chen, F., Jin, D., Meredith, P., and Rosu, G. Monitoring Oriented Programming - A Project Overview. *Proceedings of the Fourth International Conference on Intelligent Computing and Information Systems (ICICIS'09)* (2009), 72–77.

[66] Chen, F., and Rosu, G. Towards monitoring-oriented programming : A paradigm combining specification and implementation. *Electronic Notes in Theoretical Computer Science 89*, 2 (2003), 108–127.

[67] Chevassus-au Louis, B. *L'analyse des risques : l'expert, le décideur et le citoyen*. Editions Quae, 2007.

[68] CNIL. Suspension de l'utilisation des drones pour contrôler le déconfinement à Paris par le Conseil d'État : les contrôles de la CNIL. `https://www.cnil.fr/fr/suspension-de-l utilisation-des-drones-pour-controler-le-deconfinement-paris-par-le-consei l-detat-les`. Consulté le 14/09/2020.

[69] Commission européenne. Règlement d'exécution (UE) 2020/746 de la Commission. *Journal officiel de l'Union européenne* (2020).

[70] Consiglio, M., Muñoz, C., Hagen, G., Narkawicz, A., and Balachandran, S. ICAROUS Integrated Configurable Algorithms for Reliable Operations of Unmanned Systems. In *35th Digital Avionics Systems Conference (DASC), 2016 IEEE/AIAA* (2016), pp. 1–5.

[71] Corot, L. Ups va livrer des médicaments par drone à des retraités aux etats-unis. *L'Usine Digitale* (28/04/2020). Consulté le : 16/11/2020.

[72] Coste-Maniere, E., and Turro, N. The MAESTRO language and its environment : specification, validation and control of robotic missions. In *IEEE/RSJ International Conference on Intelligent Robots and Systems* (1997), pp. 836—-841.

[73] Cours des Comptes. Les drones militaires aériens : une rupture stratégique mal conduite incontournable dans les armées. Tech. Rep. I, 2020.

[74] Daly, S. Weaponized c-drones in venezuela assassination attempt. *The C-Drone Review* (08/08/2018). Consulté le : 16/11/2020.

[75] D'Angelo, B., Sankaranarayanan, S., Sánchez, C., Robinson, W., Finkbeiner, B., Sipma, H. B., Mehrotra, S., and Manna, Z. LOLA : Runtime monitoring of synchronous systems. *Proceedings of the International Workshop on Temporal Representation and Reasoning* (2005), 166–175.

[76] Danvy, O., and Filinski, A. Abstracting Control. In *ACM Conference on Lisp and Functional Programming* (1990), pp. 151–160.

[77] de Saussure, F. *Cours de linguistique générale*, payot ed. 1973.

[78] Delahaye, D. A tactic language for the system coq. In *Logic for Programming and Automated Reasoning* (2000), vol. 1955, pp. 85–95.

[79] Denney, E., Pai, G., and Johnson, M. Towards a rigorous basis for specific operations risk assessment of UAS. *AIAA/IEEE Digital Avionics Systems Conference - Proceedings 2018-September* (2018).

[80] Derler, P., Lee, E. A., and Vincentelli, A. S. Modeling Cyber – Physical Systems. In *IEEE* (2012), vol. 100, Proceedings of the IEEE, pp. 13–28.

[81] Desai, A., Ghosh, S., Seshia, S. A., Shankar, N., and Tiwari, A. SOTER : A Runtime Assurance Framework for Programming Safe Robotics Systems. *Proceedings - 49th Annual IEEE/IFIP International Conference on Dependable Systems and Networks, DSN 2019* (2019), 138–150.

[82] Desai, A., Gupta, V., Jackson, E., Qadeer, S., Rajamani, S., and Zufferey, D. P : Safe asynchronous event-driven programming. *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)* (2013), 321–331.

[83] Desai, A., Qadeer, S., and Seshia, S. A. Programming safe robotics systems : Challenges and advances. In *International Symposium on Leveraging Applications of Formal Methods* (2018), pp. 103–119.

[84] Desai, A., Saha, I., Yang, J., Qadeer, S., and Seshia, S. A. Drona : A Framework for Safe Distributed Mobile Robotics. *International Conference on Cyber-Physical Systems* (2017), 239–248.

[85] Di Ruscio, D., Malavolta, I., and Pelliccione, P. A family of domain-specific languages for specifying civilian missions of multi-robot systems. *CEUR Workshop Proceedings 1319* (2014), 16–29.

[86] Di Ruscio, D., Malavolta, I., Pelliccione, P., and Tivoli, M. Automatic generation of detailed flight plans from high-level mission descriptions. *Proceedings of the ACM/IEEE 19th International Conference on Model Driven Engineering Languages and Systems - MODELS '16* (2016), 45–55.

[87] Dijkstra, E. W. Letters to the editor : Go to statement considered harmful. *Communications of the ACM 11*, 3 (1968), 147–148.

[88] Dill, E. T., Young, S. D., and Hayhurst, K. J. SAFEGUARD : An assured safety net technology for UAS. *AIAA/IEEE Digital Avionics Systems Conference - Proceedings 2016-Decem* (2016), 1–10.

[89] Direction Générale des politiques Internes - Département thématique C : Droits des citoyens et affaires Constitutionnelles, Libertés civiles, justice, et affaires intérieures. Les conséquences de l'usage civil des drones sur la protection de la vie privée et des données à caractère personnel. Tech. rep., 2015.

[90] DJI Japan. Dji gs pro. https://www.dji.com/fr/ground-station-pro, 2016–2020.

[91] Droneii. Drone manufacturer market shares : Dji leads the way in the us. https://www.droneii.com/drone-manufacturer-market-shares-dji-leads-the-way-in-the-us. Consulté le 14/09/2020.

[92] Droneii. The drone market 2020-2025 : 5 key takeaways. https://www.droneii.com/the-drone-market-size-2020-2025-5-key-takeaways. Consulté le 14/09/2020.

[93] Droneii. Report | drone service provider ranking 2019. https://www.droneii.com/report-drone-service-provider-ranking-2019. Consulté le 14/09/2020.

[94] Droneii. The drone market environment map 2016. https://www.droneii.com/drone-market-environment-map-2016, 2016. Consulté le 14/09/2020.

[95] Dubrova, E. *Fault-Tolerant Design*. Springer New York, 2013.

[96] Eco, U. Le signe, 1988.

[97] Eriksson, S., and Lundin, M. The Drone Market in Japan. Tech. rep., 2016.

[98] Finn, R. L., and Wright, D. Privacy, data protection and ethics for civil drone practice : A survey of industry, regulators and civil society organisations. *Computer Law and Security Review 32*, 4 (2016), 577–586.

[99] Fred. Ar.drone 2 in istanbul ! https://www.helicomicro.com/2013/08/28/ar-drone-2-istanbul/, 28/08/2013. Consulté le : 16/11/2020.

[100] Frontera, G., Besada, J. A., Bernardos, A. M., Casado, E., and Lopez-Leones, J. Formal intent-based trajectory description languages. *IEEE Transactions on Intelligent Transportation Systems 15*, 4 (2014), 1550–1566.

[101] Fulton, N., Mitsch, S., Bohrer, B., and Platzer, A. Bellerophon : Tactical theorem proving for hybrid systems. *Lecture Notes in Computer Science 10499 LNCS* (2017), 207–224.

[102] Gamma, E., Helm, R., Johnson, R., and Vlissides, J. M. *Design Patterns : Elements of Reusable Object-Oriented Software*, 1 ed. Addison-Wesley Professional, 1994.

[103] Gilabert, R. V., Dill, E. T., Hayhurst, K. J., and Young, S. D. SAFEGUARD : Progress and test results for a reliable independent on-board safety net for UAS. *AIAA/IEEE Digital Avionics Systems Conference - Proceedings 2017-Septe* (2017).

[104] Gilli, A. Drones for Europe. *Briefs of the Europeran Union Institute for Security Studies*, September (2013), 1–4.

[105] Greer, C., Burns, M., Wollman, D., and Griffor, E. Cyber Physical Systems and Internet of Things in Industry. *NIST Special Publication 1900-202* (2019).

[106] Grov, G., Kissinger, A., and Lin, Y. A graphical language for proof strategies. *Lecture Notes in Computer Sciences 8312 LNCS*, 1 (2013), 324–339.

[107] Gurriet, T., and Ciarletta, L. Towards a generic and modular geofencing strategy for civilian UAVs. In *International Conference on Unmanned Aircraft Systems* (2016), pp. 540–549.

[108] Halbwachs, N., Caspi, P., Raymond, P., and Pilaud, D. The Synchronous Data Flow Programming Language LUSTRE. In *IEEE* (1991), IEEE, pp. 1305–1320.

[109] Hall, A. R., and Coyne, C. J. The political economy of drones. *Defence and Peace Economics 25*, 5 (2014), 445–460.

[110] Havelund, K., and Rosu, G. Runtime Verification - 17 Years Later. In *International Conference on Runtime Verification* (2018), pp. 3–17.

[111] Haynes, C. T., and Friedman, D. P. Engines build process abstractions. In *Proceedings of the 1984 ACM Symposium on LISP and functional programming* (1984), pp. 18–24.

[112] Heidegger, M. Sein und zeit, 1927.

[113] Henzinger, T. A. Verification of Digital and Hybrid Systems. *Verification of Digital and Hybrid Systems*, Lics 96 (2000), 1–30.

[114] Henzinger, T. A., Horowitz, B., and Majumdar, R. Rectangular hybrid games. In *International Conference on Concurrency Theory* (1999), pp. 320–335.

[115] Humphrey, L., Wolff, E., and Topcu, U. Formal Specification and Synthesis of Mission Plans for Unmanned Aerial Vehicles. In *2014 AAAI Spring Symposium Series* (2014), pp. 116–121.

[116] Huttunen, M. Civil unmanned aircraft systems and security : The European approach. *Journal of Transportation Security 12*, 3-4 (2019), 83–101.

[117] Inc., X. Xiaomi home. `https://play.google.com/store/apps/details?id=com.xiaomi.smarthome&hl=fr`, 2020-11-07.

[118] ISO/IEC JTC1. Internet of Things (IoT) Preliminary Report 2014. Tech. rep., 2015.

[119] Joint Authorities for Rulemaking of Unmanned Systems. JARUS guidelines on Specific Operations Risk Assessment (SORA). Tech. rep., 2019.

[120] Karaman, S., Rasmussen, S., Kingston, D., and Frazzoli, E. Specification and planning of UAV missions : A process algebra approach. In *American Control Conference* (2009), pp. 1442–1447.

[121] Kent, S. Model Driven Engineering. In *International Conference on Integrated Formal Methods* (2002), Springer, pp. 286—-298.

[122] Kervern, G. Y. *Eléments fondamentaux des cindyniques*. Economica, 1995.

[123] Knight, J., Xiang, J., and Sullivan, K. A Rigorous Definition of Cyber-Physical Systems. In *Trustworthy Cyber-Physical Systems Engineering*. 2016, pp. 48–73.

[124] Kong, H., He, F., Song, X., Hung, W. N. N., and Gu, M. Exponential-Condition-Based Barrier Certificate Generation for Safety Verification of Hybrid Systems. In *International Conference on Computer Aided Verification* (2013), pp. 242–257.

[125] Kühne, T. Matters of (meta-) modeling. *Software and Systems Modeling 5*, 4 (2006), 369–385.

[126] Lafaye, M., Lesage, M., and Claquin, P. Le recours aux satellites en agriculture : évolutions récentes et perspectives. *Analyse – Centre d'études et de prospective*, 67 (2014).

[127] Le Moigne, J.-L. *La théorie du système général*. 2006.

[128] Leucker, M., and Schallhart, C. A brief account of runtime verification. *Journal of Logic and Algebraic Programming 78*, 5 (2009), 293–303.

[129] López-Leonés, J., Vilaplana, M. A., Gallo, E., Navarro, F. A., and Querejeta, C. The aircraft intent description language : A key enabler for air-ground Synchronization in trajectory-based operations. *AIAA/IEEE Digital Avionics Systems Conference - Proceedings* (2007), 1–12.

[130] Lygeros, J., Godbole, D. N., and Sastry, S. A Game Theoretic Approach to Hybrid System Design. In *International Hybrid Systems Workshop* (1996), pp. 1–12.

[131] Ma, H. D. Internet of things : Objectives and scientific challenges. *Journal of Computer Science and Technology 26*, 6 (2011), 919–924.

[132] Martinet, A. Connotation, poésie et culture. *To Honor Roman Jakobson 2* (1967).

[133] Metelo, A., Braga, C., Brandão, D., and Brand, D. Towards the Modular Specification and Validation of Cyber-Physical Systems. In *International Conference on Computational Science and Its Applications* (mar 2018), pp. 80–95.

[134] Miles, T., Suarez, B., Kunzi, F., and Jackson, R. SORA Application to Large RPAS Flight Plans. *AIAA/IEEE Digital Avionics Systems Conference - Proceedings 2019-September* (2019), 1–6.

[135] Ministère de l'écologie, du développement durable, des transports et du logement. Arrêté du 11 avril 2012 relatif à la conception des aéronefs civils qui circulent sans aucune personne à bord, aux conditions de leur emploi et sur les capacités requises des personnes qui les utilisent. *Journal Officiel de la République Française*.

[136] Minsky, M. L. Matter, Minds and Models, 1968.

[137] Mitsch, S., Ghorbal, K., Vogelbacher, D., and Platzer, A. Formal Verification of Obstacle Avoidance and Navigation of Ground Robots. *The International Journal of Robotics Research 36*, 12 (2017), 027836491773354.

[138] Mitsch, S., and Platzer, A. ModelPlex : verified runtime validation of verified cyber-physical system models. *Formal Methods in System Design 49*, 1 (2016), 33–74.

[139] Molina, B. D. M., and Segarra, M. The Drone Sector in Europe. In *Ethics and Civil Drones*. Springer, 2018, pp. 7–33.

[140] Momont, A. Ambulance drone. `https://www.tudelft.nl/en/ide/research/research-labs/applied-labs/ambulance-drone/`. Consulté le : 16/11/2020.

[141] Moreau, P.-E. A choice-point library for backtrack programming. In *Implementation Technology for Programming Languages based on Logic* (1998), pp. 16–31.

[142] Moreau, P. E., Ringeissen, C., and Vittek, M. A pattern matching compiler for multiple target languages. In *Compiler Construction* (Berlin, Heidelberg, 2003), G. Hedin, Ed., Springer Berlin Heidelberg, pp. 61–76.

[143] Niedercorn, F. Avec l'AR.Drone, Parrot vise le marché des loisirs. *Les Echos* (18/01/2010). Consulté le : 14/11/2020.

[144] Nilsson, N. J. Teleo-reactive programs for agent control. *Journal of Artificial Intelligence Research 1* (1993), 139–158.

[145] Paris, T. *Modélisation de Systèmes Complexes par Composition : Une démarche hiérarchique pour la co-simulation de composants hétérogènes*. PhD thesis, 2019.

[146] Parrot SA. Freeflight pro. `https://www.parrot.com/fr/applications-et-services`, 2020-10-13.

[147] Paty, M. L'endoréférence d'une science formalisée de la nature. In *Intelligibility in science*, C. Dilworth, Ed. 1992, pp. 73–110.

[148] Perlestein, L. Les drones français, médaille d'or des meilleurs caméramens de sotchi. `https:/www.latribune.fr/technos-medias/20140218trib000815962/les-drones-francais-medaille-d-or-des-meilleurs-cameramens-de-sotchi.html`, 18/02/2014. Consulté le : 13/11/2020.

[149] Pix4D. Pix4dcapture. `https://www.pix4d.com/`, 2020-10-26.

[150] Platzer, A. Differential-algebraic Dynamic Logic for Differential-algebraic Programs. *Journal of Logic and Computation 20*, 1 (2008), 309–352.

[151] Platzer, A. Differential dynamic logic for hybrid systems. *Journal of Automated Reasoning 41*, 2 (2008), 143–189.

[152] Platzer, A. Differential Hybrid Games. *ACM Transactions on Computational Logic 18*, 3 (2017).

[153] Platzer, A. The Logical Path to Autonomous Cyber-Physical Systems. In *International Conference on Quantitative Evaluation of Systems* (2019), pp. 25–33.

[154] Platzer, A., and Quesel, J.-d. European Train Control System : A Case Study in Formal Verification. In *International Conference on Formal Engineering Methods* (2009), pp. 246–265.

[155] Politz, J. G., Martinez, A., Milano, M., Warren, S., Patterson, D., Li, J., Chitipothu, A., and Krishnamurthi, S. Python : the full monty. *ACM SIGPLAN Notices 48*, 10 (2013), 217–232.

[156] Protais, M. Pixiel perfectionne la valse des drones au puy du fou. `https://www.usinenouvelle.com/editorial/pixiel-perfectionne-la-valse-des-drones-au-puy-du-fou.N401962`, 14/07/2016. Consulté le : 13/11/2020.

[157] PwC. Clarity from above. Tech. rep., 2016.

[158] QGroundControl. Qgroundcontrol. `http://qgroundcontrol.com/`, 2020-11-04.

[159] Queinnec, C. A library of high level control operators. *ACM SIGPLAN Lisp Pointers VI*, 4 (1993), 11–26.

[160] Ramasseul, D. L'hypothèse ovni plane sur les centrales. *Paris Match* (26/02/2015). Consulté le : 16/11/2020.

[161] Rao, B., Gopi, A. G., and Maione, R. The societal impact of commercial drones. *Technology in Society 45*, May (2016), 83–90.

[162] Rassler, D. Remotely Piloted Innovation : Terrorism, Drones and Supportive Technology. Tech. rep., Combating Terrorism Center at West Point West Point United States, 2016.

[163] Rémy, P. La note du CERPA n°205 : L ' industrie aéronautique militaire serbe et la production de drones armés, 2019.

[164] Reynolds, J. C. GEDANKEN - A Simple Typeless Language Based on the Principle of Completeness and the Reference Concept. *Commun. ACM 13*, 5 (1970), 308–319.

[165] Ricketts, D., Malecha, G., Alvarez, M. M., Gowda, V., and Lerner, S. Towards Verification of Hybrid Systems in a Foundational Proof Assistant. In *International Conference on Formal Methods and Models for Codesign (MEMOCODE), 2015 ACM/IEEE* (2015), IEEE, pp. 248–257.

[166] Romanovsky, A., and Ishikawa, F. *Trustworthy cyber-physical systems engineering.* CRC Press, 2016.

[167] Sanchez-Lopez, J. L., Molina, M., Bavle, H., Sampedro, C., Suárez Fernández, R. A., and Campoy, P. A Multi-Layered Component-Based Approach for the Development of Aerial Robotic Systems : The Aerostack Framework. *Journal of Intelligent and Robotic Systems : Theory and Applications 88*, 2-4 (2017), 683–709.

[168] Santamaria, E., Pastor, E., Barrado, C., Prats, X., Royo, P., and Perez, M. Flight Plan Specification and Management for Unmanned Aircraft Systems. *Journal of Intelligent and Robotic Systems 67* (2012), 155–181.

[169] Schierman, J. D., Devore, M. D., Richards, N. D., Gandhi, N., Cooper, J. K., Horneman, K. R., Stoller, S., and Smolka, S. Runtime Assurance Framework Development for Highly Adaptive Flight Control Systems. Tech. rep., Barron Associates, Inc., 2016.

[170] Schirmer, S., Torens, C., and Adolf, F. Formal Monitoring of Risk-based Geofences. In *2018 AIAA Information Systems-AIAA Infotech@ Aerospace* (2018).

[171] Schwartz, B., Nägele, L., Angerer, A., and MacDonald, B. A. Towards a Graphical Language for Quadrotor Missions. *Workshop on Domain-Specific Languages and models for Robotic systems* (2014).

[172] SESAR Joint Undertaking. European Drones Outlook Study. Tech. rep., 2016.

[173] Sha, L. Using simplicity to control complexity. *IEEE Software 18*, 4 (2001), 20–28.

[174] Sha, L., Rajkumar, R., and Gagliardi, M. Evolving Dependable Real-Time. In *1996 IEEE Aerospace Applications Conference. Proceedings* (1996), IEEE, pp. 335–346.

[175] Simon, D., Pissard-gibollet, R., and Arias, S. Orrcad , a framework for safe robot control design and implementation. In *1st National Workshop on Control Architectures of Robots : software approaches and issues* (2006).

[176] Sirigineedi, G., Tsourdos, A., White, B. A., and Zbikowski, R. Kripke modelling and verification of temporal specifications of a multiple UAV system. *Annal of Mathematics and Artificial Intelligence 63*, 1 (2011), 31–52.

[177] Sitaram, D., and Felleisen, M. Control Delimiters and Their Hierarchies. *Lisp and symbolic computation 3*, 1 (1990), 67–99.

[178] Sogokon, A., Ghorbal, K., Tan, Y. K., and Platzer, A. Vector Barrier Certificates and Comparison Systems. In *International Symposium on Formal Methods* (2018), pp. 418–437.

[179] Soler, L. Qu'est-ce qu'un modèle scientifique ? Des caractéristiques du modèle qui importent du point de vue de l'enseignement intégré de science et de technologie. *Spirale. Revue de recherches en éducation 52*, 1 (2013), 177–214.

[180] Stachowiak, H. *Allgemeine modelltheorie*. Springer, 1973.

[181] Stachowiak, H. *La pensée scientifique*. 1978, ch. Les modèles, pp. 155–177.

[182] Stojmenovic, I., and Zhang, F. Inaugural issue of 'cyber-physical systems'. *Cyber-Physical Systems 1*, 1 (2015), 1–4.

[183] Strachey, C., and Wadsworth, C. P. Continuations : A Mathematical Semantics for Handling Full Jumps. *Higher-Order and Symbolic Computation 13* (2000), 135–152.

[184] Su, J., He, J., Cheng, P., and Chen, J. A Stealthy GPS Spoofing Strategy for Manipulating the Trajectory of an Unmanned Aerial Vehicle. *IFAC-PapersOnLine 49*, 22 (2016), 291–296.

[185] Su, W., Abrial, J. R., and Zhu, H. *Formalizing hybrid systems with Event-B and the Rodin Platform*, vol. 94. Elsevier B.V., 2014.

[186] SymPy Development Team. Sympy. `https://www.sympy.org/en/index.html`.

[187] Terkildsen, K. H., and Jensen, K. Towards a tool for assessing UAS compliance with the JARUS SORA guidelines. *2019 International Conference on Unmanned Aircraft Systems, ICUAS 2019* (2019), 460–466.

[188] Torens, C., Durak, U., Nikodem, F., and Schirmer, S. Formally Bounding UAS Behavior to Concept of Operation with Operation-Specific Scenario Description Language. In *AIAA Scitech 2019 Forum* (2019).

[189] Van de Voorde, P., Gautama, S., Momont, A., Ionescu, C. M., De Paepe, P., and Fraeyman, N. The drone ambulance [A-UAS] : golden bullet or just a blank ? *Resuscitation 116* (2017), 46–48.

[190] Vandenberghe, F. Introduction à la sociologie (cosmo) politique du risque d'Ulrich Beck. *Revue du MAUSS 1* (2001), 25–39.

[191] Viard, L., Ciarletta, L., and Moreau, P.-E. Monitor-Centric Mission Definition With Sophrosyne. In *2019 International Conference on Unmanned Aircraft Systems (ICUAS)* (2019), pp. 111–119.

[192] Viard, L., Ciarletta, L., and Moreau, P.-E. A Mission Definition, Verification and Validation Architecture. In *Formal Methods. FM 2019 International Workshops*, vol. 12232. Aug. 2020, pp. 281–287.

[193] Wittgenstein, L., Klossowski, P., and D'A, P. Tractatus logico-philosophicus, suivi de investigations philosophiques.

[194] Yang, J., Islam, M. A., Murthy, A., Smolka, S. A., and Stoller, S. D. A simplex architecture for hybrid systems using barrier certificates. In *International Conference on Computer Safety, Reliability, and Security* (2017), pp. 117–131.

Résumé

La conception de systèmes cyber-physiques est une discipline émergente à l'interface de nombreux domaines d'ingénierie. Ces systèmes se caractérisent notamment par une identité double, liant le monde des contrôleurs, discret, à celui du matériel, continu. Les errements d'un contrôleur, qu'ils soient dus à un programme erroné ou à la manifestation d'un aléa de l'environnement, sont susceptibles de produire des conséquences désastreuses. Une attention particulière doit donc être apportée à leur programmation. Le travail présenté dans cet ouvrage est une réponse à ce défi.

Nous proposons un langage dédié à la programmation des systèmes cyber-physiques, `Sophrosyne`, ainsi qu'une méthode formelle de vérification des missions résultantes. Le langage repose sur des structures de supervision, permettant au système d'adapter son comportement selon la survenance d'aléas. Il présente de plus un volet de modélisation continue du système au moyen d'équations différentielles, duquel dérive la vérification formelle des missions exprimée en logique dynamique différentielle. Divers outils ont été développés autour de `Sophrosyne` pour assurer la planification, la compilation, l'analyse, et l'exécution de missions. Ils constituent une chaîne logicielle complète allant d'une interface graphique assistant la conception de mission jusqu'à son exécution sur le système réel. Ces outils ont été mis en oeuvre sur des projets d'inspections aériennes d'infrastructures par drone. Les travaux présentés sont illustrés par ces applications drones.

Mots-clés: Systèmes cyber-physiques, Programmation de mission, Langages dédiés, Preuve de théorème

Abstract

Building cyber-physical systems is an up-and-coming discipline which involves many engineering domains. Cyber-physical systems have a controller monitoring their physical behaviour, resulting in intertwined discrete and continuous evolution. Faulty programs or environmental hazards might lead to unwanted control and disastrous consequences. Safe operation of cyber-physical systems requires to pay dedicated attention to their programming. Our work attempts to provide a solution to this challenge.

We present a domain specific language for programming cyber-physical systems, `Sophrosyne`, as well as a formal method to verify the correction of the resulting missions. The language is based on monitoring control structures, which provide reactive behaviours to the system. It furthermore includes continuous modelling of the system with differential equations to enable verification of missions using differential dynamic logic. Various softwares have been built to provide `Sophrosyne` with mission plannification, compilation, analysis, and execution. Together they form a complete toolchain from a graphical user interface supporting the definition of a mission to its execution on the real system. These tools have been used to define aerial inspections of infrastructure with unmanned aircraft. We demonstrate our contribution on such applications.

Keywords: Cyber-Physical Systems, Mission definition, Domain Specific Language, Theorem Proving