AMSA, un framework dédié à la simulation des lois de contrôle pour des voiliers de compétition

Emilien Lavigne

THÈSE DE DOCTORAT EN INFORMATIQUE

UNIVERSITÉ
DE BRETAGNE OCCIDENTALE
COMUE UNIVERSITÉ BRETAGNE LOIRE

ÉCOLE DOCTORALE N° 601
Mathématiques et Sciences et Technologies de l'Information et de la Communication
Spécialité : *Informatique*

Par
Émilien LAVIGNE

AMSA, un framework dédié à la simulation de lois de contrôle pour des voiliers de compétition

Thèse présentée et soutenue à Brest, le 05 décembre 2019
Unité de recherche : UMR 6285 Lab-STICC

Rapporteurs avant soutenance :

Sébastien GERARD	Directeur de recherche, HDR, CEA List
Charles LESIRE-CABANIOLS	Directeur de recherche, HDR, ONERA

Composition du Jury :

Président :	Jean-Marc JEZEQUEL	Professeur, IRISA
Examinateurs :	Karen GODARY-DEJEAN	Maître de conférences, Université de Montpellier
	Sébastien GERARD	Directeur de recherche, HDR, CEA List
	Charles LESIRE-CABANIOLS	Directeur de recherche, HDR, ONERA
	Frank SINGHOFF	Professeur, Université de Bretagne Occidentale
Dir. de thèse :	Jean-Philippe BABAU	Professeur, Université de Bretagne Occidentale

Invités

Co-encadrant :	Goulven GUILLOU	Maître de conférences, Université de Bretagne Occidentale
	Antoine GAUTIER	Directeur des études, MerConcept

Remerciements

Il me paraît important en premier lieu de remercier **François Gabart**, qui a été à l'origine de ce travail de thèse et qui m'a accueilli dans sa structure *MerConcept*. Les projets qu'il porte avec dynamisme sont source d'inspiration, et je le remercie de m'avoir offert l'opportunité d'y prendre part.

Ce projet n'aurait pas pu voir le jour sans mes deux encadrants, **Jean-Philippe Babau** et **Goulven Guillou**. Je tiens à les remercier particulièrement pour la disponibilité qu'ils ont eue ces trois dernières années, pour la pédagogie dont ils ont fait preuve pour accompagner les premiers pas d'un ingénieur dans le monde de la recherche, et pour la patience qu'ils ont montrée les nombreuses fois où ils m'ont entendu dire « Désolé, je suis en retard, j'étais sur l'eau... »

Je tiens également à remercier **Sébastien Gérard** et **Charles Lesire-Cabaniols**, qui ont accepté de rapporter mon travail, ainsi que **Karen Godary-Dejean**, **Jean-Marc Jézéquel** et **Frank Singhoff**, qui complètent le jury de cette thèse.

J'ai passé la majeure partie de ces trois dernières années au sein de *MerConcept*, et il me semble primordial de remercier l'équipe dans son ensemble, pour avoir su si bien m'entourer. Vue notre récente et spectaculaire expansion, il me parait difficile de citer l'ensemble de l'équipe ici. Toutefois, je tiens tout de même à remercier **Antoine**, pour la confiance et l'autonomie qu'il m'a toujours accordées (et pour les navigations en open) ; **Vivien**, pour les bons moments passés ensemble et pour m'avoir appris à utiliser *le bon outil* ; **Guillaume**, pour son soutient indéfectible dans mes projets les plus fous ; **Benoît**, pour m'avoir initié aux technologies vraiment pratiques dans le vaste monde de l'électronique embarquée ; **Erwan**, pour m'avoir fait découvrir le monde de l'hydraulique (et les risques qui l'entourent) mais aussi la science de l'isolation thermique ; **Fred**, pour l'intérêt qu'il porte à mes travaux ; **Steph**, pour ses conseils avisés tant en matière de mécanique que de mustélidés ; **Luis**, pour ses explications toujours riches et pour son accueil chaleureux ; **Antoine**, pour ses bons tuyaux sur la vie concarnoise et forestienne ; **Isa**, pour toutes les blagues de haut niveau que l'on s'est échangées ; **Seb**, pour les quarts passés en sa compagnie à découvrir le fonctionnement d'un géant des mer ; **Nico**, pour m'avoir appris que rapidité et qualité ne sont pas incompatibles ; **Thomas**, pour m'avoir initié au monde du composite ; **Corentin**, pour m'avoir transmis le plaisir du travail bien fait ; **Pascal**, pour sa bonne humeur communicative ; **Armelle**, pour la bienveillance dont elle nous entoure au quotidien ; **Tiphaine**, pour l'énergie qu'elle a injectée dans ce projet ; et **tous les autres**, qui font aujourd'hui de MerConcept l'équipe de rêve pour s'épanouir pleinement !

Mes remerciements vont aussi aux thésards du Lab-STICC avec qui j'ai eu le plaisir de partager un bureau au labo, en particulier **Hamza** et **Christophe**, pour les échanges constructifs autour de nos travaux respectifs. Je tiens également à citer les enseignants du département informatique, pour la bonne ambiance dans laquelle ils m'ont accueilli.

Je remercie également la société *Madintec* qui m'a facilité l'accès à son pilote afin de pouvoir l'inclure dans les simulations.

Enfin, un grand merci à tous ceux qui m'ont soutenu, encouragé et motivé durant ces trois dernières années, en particulier ma sœur **Valérie** pour le travail de relecture qu'elle a fourni sur ce manuscrit, mais également mes parents, ma famille, mes proches, et tous ceux que je n'ai pas encore cités dans ces lignes. Ils se reconnaîtront !

Table des matières

Chapitre 1

Introduction

1.1 Contexte de l'approche

Cette thèse est le fruit d'une collaboration entre l'écurie de course au large *MerConcept* et le laboratoire *Lab-STICC*, autour d'un projet sur le pilotage automatique des voiliers de compétition. *MerConcept* est aujourd'hui en charge de l'exploitation de plusieurs bateaux, dont le trimaran MACIF, multicoque océanique de la classe Ultime, le monocoque IMOCA 60 APIVIA et deux Figaro BENETEAU 3, auxquels s'ajoutent les bateaux actuellement en construction, dont un nouvel Ultime. Principalement centrée autour des grands rendez-vous de la navigation en solitaire (record et course autour du monde, Vendée-Globe, Route du rhum, Solitaire du Figaro...), l'équipe est en permanente quête de performances sportives et d'améliorations technologiques.

Dans le contexte de la navigation en solitaire, le pilote automatique est un élément indispensable. Il permet au marin de se libérer afin de pouvoir régler les voiles, préparer sa navigation, étudier la météo, manger, dormir... A l'échelle d'une course océanique ou d'un tour du monde, le pilote est intensivement utilisé. Bien que cela ne soit pas exactement quantifié, les skippers de *MerConcept* estiment qu'ils barrent seulement entre 5% et 20% du temps lors d'une course de plusieurs jours. Le pilote automatique joue donc un rôle majeur dans la fiabilité et la performance du voilier.

Les pilotes utilisés aujourd'hui sur les bateaux de course au large sont souvent des solutions commercialisées sous forme de boîte noire ou grise : l'utilisateur n'a pas un accès direct à la loi de contrôle, mais il peut éventuellement paramétrer cette dernière en choisissant la valeur de certains coefficients. Le nombre de coefficients accessibles varie fortement en fonction du pilote considéré, de deux ou trois pour les modèles les plus simples à plusieurs centaines pour les solutions les plus complexes et les plus ouvertes. Les solutions « paramétrables » sont les plus prisées pour pouvoir s'adapter aux situations rencontrées et aux caractéristiques du bateau.

Cette possibilité de personnalisation engendre une complexité dans le réglage, avec une combinatoire très élevée. Le paramétrage d'un pilote, tout comme la mise au point d'une nouvelle loi de contrôle, se révèle être un exercice délicat, d'autant que le paramétrage optimal varie en fonction des situations de navigation. Par ailleurs, les évolutions récentes des bateaux ont amené de nouveaux modes de navigation. L'arrivée de foils dans la course au large permet de faire « voler » les bateaux au-dessus de la surface (*cf.* figure 1.1), ce qui nécessite de diversifier et d'enrichir les lois de contrôle. Dans ce contexte, ce travail de thèse s'intéresse à la mise au point, via la simulation, d'un pilotage fiable et performant pour des voiliers de compétition.

FIGURE 1.1 – Le Trimaran MACIF en navigation lors d'un entraînement

1.2 Problématiques

La mise en place du contrôle d'un système physique est une problématique classique relevant de l'automatique, qui a été de nombreuses fois étudiée. Cependant, un voilier de course possède des spécificités qui imposent certaines contraintes sur le contrôle. D'une part, il s'agit d'un système fortement dépendant de son environnement (en particulier le vent et l'état de mer). D'autre part la régulation ne peut intervenir que sur la barre, pour des questions réglementaires. Ainsi, un seul degré de liberté est directement contrôlable, ce qui complique notablement les politiques d'asservissement. En particulier, les corrections demandées à la trajectoire sont parfois supérieures aux possibilités physiques du système de barre, le bateau devient alors incontrôlable. On parle souvent de départ au lof ou à l'abattée : le bateau adopte une gîte anormalement élevée, le safran sort partiellement de l'eau, réduisant encore sa contrôlabilité. Dans le cas des multicoques, une telle situation conduit souvent au retournement du bateau (chavirage), ce qui peut occasionner sa perte totale. Le barreur humain, connaissant les performances et les caractéristiques de son bateau, va éviter de telles situations en anticipant les conditions qui pourraient y mener. Ainsi, comme le synthétisent les travaux de G. Guillou [Gui10], le pilotage ne consiste pas à essayer de contrecarrer l'ensemble des forces qui s'appliquent sur le voilier, mais plutôt à composer avec elles, parfois dans une stratégie de sauvegarde.

Qu'il s'agisse du développement d'un nouveau contrôleur ou du paramétrage d'un pilote existant, l'obtention d'une politique de contrôle qui respecte l'ensemble des contraintes de la navigation au large est une tâche ardue. La simulation du comportement du voilier dans son environnement semble indispensable pour mettre au point et tester les performances du contrôle. Cela est d'autant plus vrai que les grands voiliers de compétition passent plus du tiers de leur temps en chantier d'optimisation, et requièrent une logistique lourde pour leur mise en œuvre. Les navigations consacrées au développement et aux essais en mer du pilote automatique sont rares et coûtent cher. La simulation doit permettre de s'affranchir de ces contraintes, au moins partiellement. Elle constitue un point de passage obligé dans la mise au point des lois de contrôle.

Le développement d'un simulateur *ad-hoc* pour un bateau spécifique évoluant dans son environnement reste un travail délicat. Actuellement, les modèles de bateaux et d'environnement sont développés par des experts des domaines concernés, à savoir les architectes navals et les mécaniciens des fluides, tandis que les automaticiens se chargent des lois de contrôle. Ces approches mènent à des solutions centrées sur le code métier, difficilement intégrables, et qui n'ont en général pas été pensées dans l'optique de simulations globales. Les enjeux de la simulation peuvent êtres classés selon quatre axes principaux :

la performance qui concerne l'optimisation du code métier ou la mise en place d'exécutions parallèles et concurrentes de divers modules ;

la précision qui dépend de la qualité du code métier et des modèles utilisés ;

la facilité d'intégration de composants hétérogènes (code métier dans divers langages de programmation, contrôleur existant) ;

la facilité d'exploitation du simulateur.

Les problèmes de performance et de précision relèvent de l'optimisation de chaque composant du simulateur. Ils sont liés aux plateformes support (logicielles et matérielles) et à l'optimisation du code. Des approches comme HLA/RTI [Dah97 ; DM98] sont centrées sur ces problématiques.

Ce travail de thèse s'intéresse quant à lui à la facilité d'intégration et d'exploitation. Ces deux problématiques relèvent du génie logiciel : il s'agit de développer des solutions de simulation qui permettent d'intégrer des éléments hétérogènes au sein d'un même simulateur, tout en gardant la possibilité d'interchanger ces éléments. Il paraît également important de faciliter le paramétrage du simulateur, tout en permettant la réutilisation de briques logicielles existantes, afin de faciliter la conduite de simulations et l'exploitation des résultats produits.

L'intégration du code métier spécifique est une tâche complexe. Les différents composants d'un simulateur de voilier présentent une forte hétérogénéité, par la nature même du modèle qu'ils utilisent, la nature et le format des données qu'ils échangent et la manière dont ils doivent être exécutés. Par exemple, en ce qui concerne les modèles de bateaux, on trouve des modèles mathématiques relativement simples qui fonctionnent avec peu de degrés de liberté [Mel15 ; XJ13 ; Jau+12 ; Jen10], des modèles physiques très complexes faisant intervenir de lourds calculs en mécanique des fluides [Lom+12 ; Bin+08] ou encore des modèles uniquement basés sur les données acquises en situation réelle [Lav+16]. L'intégration de ces différents modèles de bateaux ou d'environnement passe souvent par la réécriture d'une large partie du logiciel.

D'autre part, en fonction des cas d'étude et des objectifs de la simulation, il est nécessaire de pouvoir paramétrer différents scénarios de simulation et de pouvoir les exécuter facilement, afin de produire des résultats sous une forme facilement analysable. L'utilisation d'un simulateur ne doit pas requérir une connaissance approfondie de son fonctionnement interne. Il doit être facilement exploitable par les personnes en charge du développement et de la mise au point des lois de contrôle. La configuration de la simulation pour explorer des situations diverses (vent, configuration du bateau, configuration du contrôleur, erreurs d'acquisition, problèmes de barre...) doit être facilitée.

1.3 Contribution

Les réponses à ces défis requièrent la mise en place d'une architecture logicielle qui définit l'organisation des différents composants du simulateur. Ce travail propose un cadre logiciel pour une telle architecture, le framework AMSA, assorti d'un style architectural répondant aux problématiques d'adaptation générées par l'hétérogénéité des composants d'un simulateur. Les principales contributions de cette thèse sont :

La modélisation des constituants d'un simulateur de voilier. Le framework AMSA propose, au travers de quatre méta-modèles, de modéliser les composants physiques et logiciels d'un simulateur de voilier évoluant dans son environnement.

La standardisation des interfaces de communication permettant aux deux constituants principaux du simulateur, le modèle de bateau et le modèle d'environnement, d'échanger des données de manière standardisée. Au cœur du style architectural AMSA, ces interfaces permettent de gérer l'hétérogénéité des divers modèles utilisés.

L'exécution d'une simulation, à travers la modélisation des événements qui interviennent avant, pendant et après une simulation. Un mécanisme de scénario permet de gérer ces événements, et ainsi de paramétrer le simulateur à loisir pour obtenir des résultats facilement exploitables en fonction des objectifs de simulation.

1.4 Organisation du document

La suite de ce document est décomposée en trois parties, qui correspondent respectivement à l'état de l'art, la proposition et l'évaluation du travail réalisé durant cette thèse. Chaque partie est constituée de plusieurs chapitres en fonction des thématiques abordées.

La première partie dresse un état de l'art des solutions existantes. Elle est constituée de quatre chapitres. Le chapitre 2 expose les principaux éléments de contexte, en abordant les notions propres au voilier et à son environnement ainsi qu'aux parties matérielles et logicielles des pilotes automatiques existants. Le chapitre 3 présente les principes et les intérêts de l'*Ingénierie Dirigée par les Modèles* et motive son utilisation pour la mise en place d'une architecture logicielle. Le chapitre 4 présente les différentes problématiques qui interviennent lors de la définition d'un système à composants, ainsi que les solutions qui y sont couramment apportées pour la modélisation des systèmes cyber-physiques. Enfin, le chapitre 5 propose un tour d'horizon des architectures logicielles existantes pour le contrôle et la simulation de systèmes cyber-physiques.

La seconde partie, qui détaille la proposition faite dans cette thèse, se compose de cinq chapitres. Le chapitre 6 présente les principes généraux du framework AMSA et les choix qui ont été faits en réponse aux questions soulevées dans la première partie. Le chapitre 7 détaille la manière dont un système à composants est modélisé puis mis en œuvre au sein du framework. Le chapitre 8 expose le style architectural mis en place, en expliquant comment celui-ci permet de répondre efficacement aux problèmes d'intégration de composants hétérogènes. Les mécanismes de scénarios, qui permettent de paramétrer, d'exécuter et d'exploiter les résultats des simulations sont exposés au chapitre 9. Enfin, le chapitre 10 présente l'ensemble des outils du framework qui permettent de manière concrète de modéliser, de générer puis d'exécuter des simulateurs de voiliers de compétition.

La troisième partie apporte des éléments pour l'évaluation de la proposition. Elle se compose du chapitre 11 qui présente les différents modèles de vent, de bateaux et de pilotes qui ont été intégrés au sein du framework sous forme de composants hétérogènes, et du chapitre 12 qui expose et commente les résultats de diverses simulations, en expliquant comment elles ont permis la mise au point de lois de contrôle.

Enfin, un chapitre de conclusion propose une discussion sur le travail effectué, tout en indiquant les pistes envisagées pour la poursuite de ce travail.

Première partie

État de l'art

Chapitre 2

Contexte du travail

La voile est utilisée comme moyen de propulsion sur l'eau depuis plusieurs millénaires. On trouve des voiliers de formes diverses, construits pour des objectifs variés. Au cours du siècle dernier, l'émergence d'une discipline nouvelle, la course au large, a entraîné l'apparition de nouveaux types de navires, taillés pour la vitesse et entièrement conçus autour de la notion de performance. Dans la suite de ce travail, les termes *voiler de compétition* et plus simplement *voilier* désignent ces engins de course, extrêmes par leurs caractéristiques et leurs dimensions.

L'univers de la mer dispose d'un vocabulaire spécifique, qui peut sembler opaque au néophyte. L'objectif de ce chapitre est d'introduire les éléments nécessaires à la compréhension des trois principaux éléments qui interviennent dans le contrôle d'un voilier de compétition, à savoir le voilier, son environnement, et l'ensemble des matériels et logiciels qui permettent le contrôle. Le vocabulaire et les notions propres à chaque domaine sont introduits, et des éléments de modélisation basés sur de précédents travaux sont présentés.

2.1 Le voilier de course

2.1.1 Présentation générale

Il existe de nombreux types de voiliers, de toutes dimensions. Il est possible de les catégoriser selon plusieurs paramètres comme la longueur, le nombre de coques, le nombre de mâts ou encore la surface de voile. Ce travail s'intéresse particulièrement aux voiliers de compétition. De tels engins sont entièrement orientés vers la performance dès leur conception : les matériaux employés, les études réalisées et les processus de fabrication en font des engins hors-norme.

Comme on le voit sur la figure 2.1, un voilier est constitué d'une ou plusieurs coques, d'un ou plusieurs mâts qui portent des voiles et d'appendices implantés sous les coques qui interagissent avec l'eau. On parle de monocoque si le voilier dispose d'une seule coque, et de multicoque dans le cas inverse (on réserve les termes de catamaran aux voiliers à deux coques, et de trimaran aux voiliers à trois coques).

Les voiles, portées par le mât (généralement unique sur les voiliers modernes) permettent de créer une force de propulsion. Cette force peut être ajustée en modifiant l'incidence de la voile au moyen de cordages nommés écoutes. L'incidence optimale dépend de l'angle que forme le vent avec l'axe du bateau, il est donc primordial de régler les voiles à chaque modification de cet angle (modification de la direction ou de la vitesse du bateau, changement de vent). La voile principale, nommée *grand-voile*, est située en arrière du mât. Son action est complétée par une voile d'avant, dont on peut modifier la taille en fonction des conditions météorologiques rencontrées et de l'angle au vent.

Les appendices ont divers objectifs selon leur nature : les safrans permettent de diriger le bateau, les dérives, comme leur nom l'indique, s'opposent à la dérive du bateau (mouvement de dérapage latéral), tandis que les foils génèrent une portance verticale qui permet de soulever le bateau pour le faire voler. Cette thèse s'intéresse aux pilotes automatiques qui agissent sur les safrans.

FIGURE 2.1 – Anatomie du trimaran MACIF : 30 m de long par 23 m de large pour un tirant d'air de 35 m

2.1.2 Vocabulaire

Les allures

La marche d'un voilier dépend principalement de l'angle que forme son axe avec le lit du vent. En fonction de cet angle, on répertorie différentes allures présentées sur la figure 2.2. Bien que la définition exacte des allures dépende du support utilisé et soit propre à chaque marin, les termes de près et portant désignent généralement les allures de navigation comprises respectivement dans les intervalles $[30^\circ\ ;\ 70^\circ]$ et $[120^\circ\ ;\ 150^\circ]$ par rapport au vent réel. Entre les deux, on parle de travers (ou *reaching*). Les allures sont symétriques par rapport à l'axe du vent. Lorsque que le bateau reçoit le vent par son côté droit, on dit qu'il est tribord amure, bâbord amure dans le cas inverse.

Le bateau est face au vent quand l'angle au vent réel est inférieur à 30°. Il est impossible de progresser dans cette zone, et la remontée au vent doit se faire en tirant des bords, on parle de louvoyage. La route en vent arrière (progression le long de l'axe du vent) est toujours possible, mais le bateau est limité à la vitesse du vent, ce qui rend cette allure peu efficace. Aussi, il est courant de tirer des bords également lorsque l'on cherche à descendre dans le lit du vent.

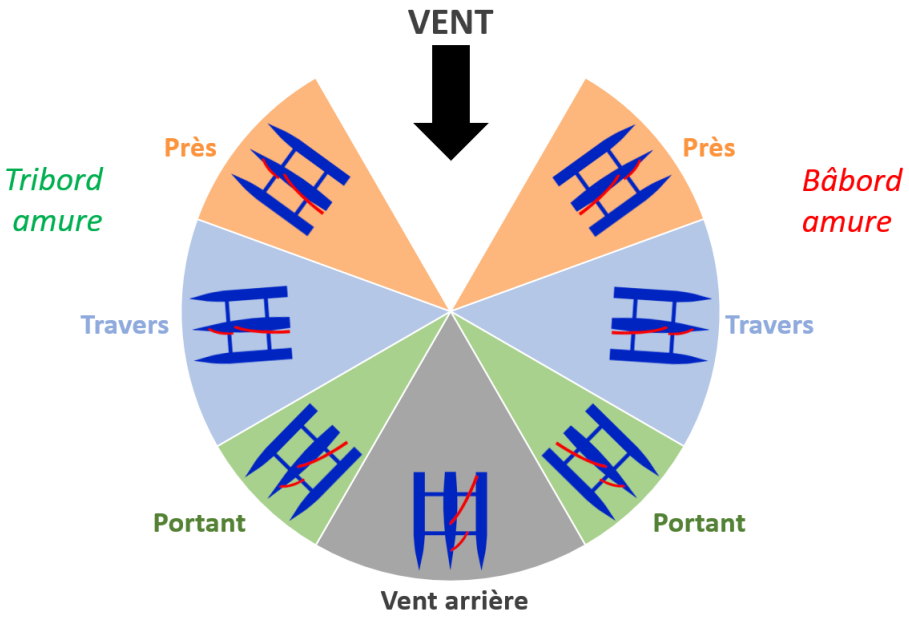

FIGURE 2.2 – Les allures

L'attitude de la plateforme

La plateforme désigne la coque du bateau dans le cas d'un monocoque, et l'assemblage des coques et des bras de liaison dans le cas des multicoques. L'attitude est la caractérisation des mouvements et de la position de la plateforme dans les trois dimensions. Il existe plusieurs conventions pour décrire l'attitude : certaines sont utilisées par les marins, d'autres par les architectes et les ingénieurs, d'autres encore sont empruntées au monde de l'aéronautique. Cette multiplicité de conventions peut poser problème lors de l'intégration de différents modèles dans un même simulateur, comme cela est détaillé au chapitre 8.

Le but de cette partie n'est pas de décrire l'ensemble des conventions existantes, mais simplement d'introduire le vocabulaire utilisé dans ce travail. L'attitude se compose des mouvements linéaires et angulaires du bateau. Nous indiquons ici les termes français, ainsi que leurs équivalents en anglais qui seront utilisés dans les modèles. Comme le montre la figure 2.3, les trois mouvements linéaires sont le cavalement (*surge*), l'embardée (*sway*) et le pilonnement (*heave*), respectivement dans l'axe longitudinal, transversal et vertical. Les rotations autour de ces trois axes sont le roulis (*roll*), le tangage (*pitch*) et le lacet(*yaw*). Cependant, l'angle de roulis est communément appelé gîte (*heel*).

Route surface et route fond

Deux référentiels peuvent être utilisés pour décrire la vitesse du bateau : l'un lié à la surface du plan d'eau, l'autre lié au fond. La vitesse par rapport à l'eau, mesurée à l'aide d'un speedomètre, est notée SOW (*Speed Over Water*), tandis que la vitesse fond, mesurée par un GPS, est notée SOG (*Speed Over Ground*).

Le GPS permet également d'obtenir la route sur le fond (COG, *Course Over Ground*), qui correspond à l'angle entre la trajectoire du bateau et le nord géographique. Le compas permet quant à lui d'obtenir le cap (souvent désigné par l'abréviation Hdg, pour *heading*), à savoir l'angle entre l'axe du bateau et le nord magnétique. La déclinaison magnétique doit être prise en compte pour corriger le cap magnétique et obtenir le cap géographique. Certaines centrales inertielles modernes fournissent directement le cap géographique, ce qui permet de s'affranchir de cette contrainte.

La différence entre le *heading* et le COG est due au courant et à la dérive (*leeway*) du bateau qui a tendance à déraper latéralement.

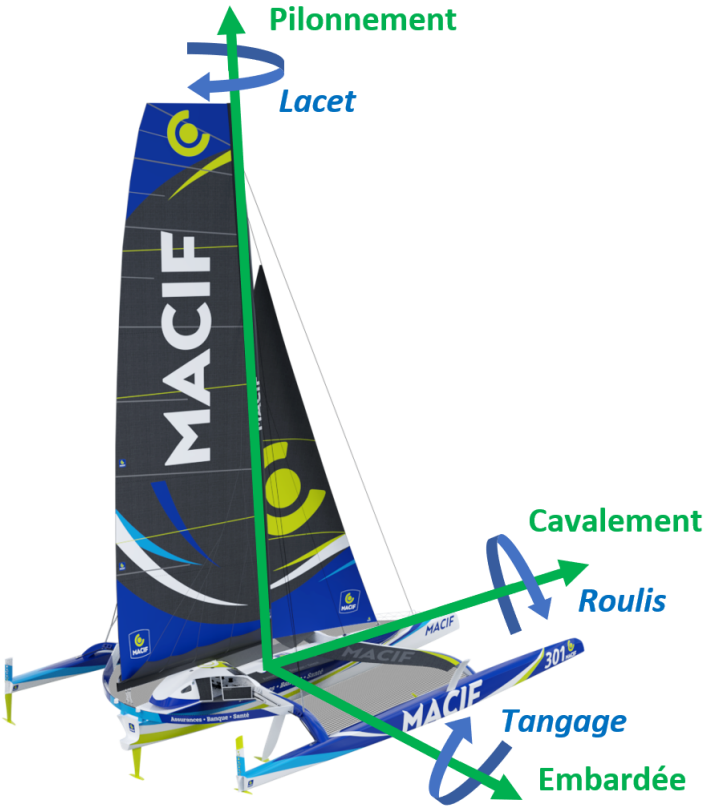

FIGURE 2.3 – La description de l'attitude de la plateforme

2.1.3 Modélisation du voilier

La modélisation du voilier et la simulation de son comportement peuvent avoir plusieurs objectifs. Cette partie propose un aperçu des différentes approches connues en fonction du contexte et des attentes de la modélisation.

Un des objectifs de la simulation est de recréer un environnement réaliste pour permettre à un utilisateur de simuler une situation de navigation [MV13]. On parle alors de simulateur physique. Un tel simulateur peut être utilisé pour de multiples objectifs [Moo+09] : entraînement, enseignement, découverte, pratique. Selon le contexte applicatif, la qualité du modèle physique du bateau doit être adaptée. Les résultats produits par un simulateur peuvent être comparés à des données acquises durant les navigations pour en évaluer la pertinence [Cla14].

Dans d'autres cas, la mise en situation de l'utilisateur n'est pas nécessaire, voire impossible (par exemple lorsque la simulation est menée en temps accéléré). Dans ces cas, le simulateur se réduit souvent au modèle mécanique du voilier. Du point de vue du mécanicien, le voilier est un mobile évoluant entre deux environnements fluides séparés par une surface libre. Les efforts auxquels il est soumis dépendent de nombreux phénomènes complexes, fortement non-linéaires, qui sont liés à l'environnement dans lequel il évolue.

Une étude des différents efforts aérodynamiques et hydrodynamiques permet de calculer les accélérations de la plateforme, et, par intégration, d'en obtenir la vitesse et la position. Une telle résolution mécanique nécessite cependant une bonne connaissance des efforts, afin de réussir à équilibrer le voilier (éviter les rotations parasites). Une des possibilités est de limiter le nombre de degrés de liberté (DDL) qui sont calculés : on trouve ainsi certains modèles qui ne prennent en compte que 3 ou 4 DDL sur les 6 disponibles [Jau+12 ; XJ13 ; Mel15]. Il est également envisageable de résoudre les équations de la dynamique des fluides pour estimer ces efforts [NCN17], ce qui peut être utile pour optimiser les formes de coques et de voiles [Lom+12], ou encore pour étudier l'interaction entre plusieurs bateaux [Spe+10].

Cependant, dans la majorité des cas, les efforts sont estimés de manière simple en introduisant un coefficient de portance C_l pour les voiles et un coefficient de traînée C_d pour les coques. Ces coefficients, multipliés par la surface de contact et le carré de la vitesse de l'écoulement, donnent une approximation de la résultante des efforts. La résolution des équations de la dynamique du voilier implique de disposer des inerties et des amortissements caractéristiques du bateau. Ceux-ci peuvent être estimés analytiquement (à partir des plans de conception), par simulation numérique, ou encore par une série d'essais en mer qui permettent d'identifier les différentes valeurs des coefficients [Bin+08].

La simulation à l'aide de modèles mécaniques de voiliers plus ou moins simplifiés a de nombreux domaines d'application. Les plus fréquemment rencontrés sont l'évitement d'obstacle [Jen10 ; Ste+10], l'aide à la prise de décision stratégique [Bin+08] ainsi que la mise au point de lois de contrôle pour le pilotage [Tom10 ; Gui10 ; XJ13 ; Mel15] et le réglage [Jau04] de voiliers.

2.2 L'environnement

Une des spécificités des systèmes cyber-physiques est l'influence prédominante de l'environnement sur leur comportement. La simulation du système requiert donc de disposer d'une vue de l'environnement, ainsi que d'un modèle d'interactions entre le système et l'environnement. On désigne ici par *environnement* tous les paramètres et les facteurs extérieurs au système étudié, et qui interagissent avec lui de manière non négligeable. Cette interaction peut être directe ou non, et elle peut être volontaire ou subie. Cela dépend à la fois de l'environnement, mais également de la nature du système. Ainsi un voilier pourra utiliser une rafale de vent pour assurer sa propulsion (interaction volontaire), tandis que cette même rafale apparaîtra comme une perturbation pour un avion (interaction subie).

Dans le cas d'un voilier de course en haute mer, les éléments d'interaction sont principalement le vent et la surface de la mer. Cette partie présente la vision usuelle de ces deux éléments par les utilisateurs du domaine, elle fournit également quelques éléments pour leur modélisation.

2.2.1 Le vent

Vent apparent et vent réel

Le vent constitue la principale source d'énergie pour le voilier. Sa mesure et sa caractérisation sont essentielles pour permettre l'utilisation du plein potentiel du bateau. Classiquement, on distingue le vent perçu par le bateau (vent apparent) et le vent établi sur le plan d'eau tel qu'il serait perçu par un observateur immobile (vent réel). Comme on peut le constater sur la figure 2.4, la vitesse du bateau crée une composante de vent vitesse qui, ajoutée au vent réel, vient modifier la force et la direction du vent apparent.

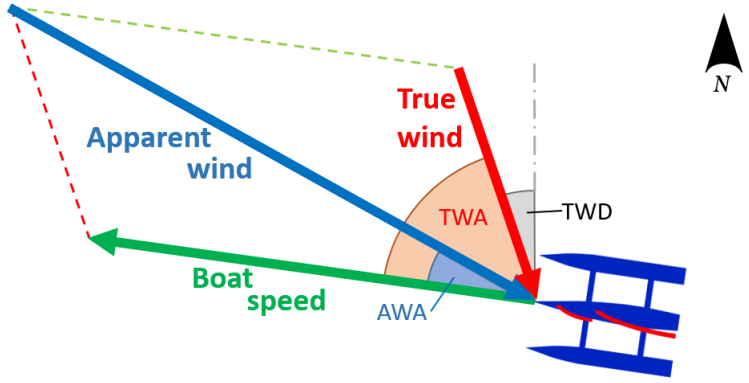

FIGURE 2.4 – Le triangle de vent : le vent apparent est obtenu en sommant le vent réel et le vent-vitesse, lui-même opposé à la vitesse du bateau

Notons que le vecteur vitesse représenté sur la figure est représentatif de la vitesse du bateau sur le fond (SOG). A cause des phénomènes de courant et de dérive, il n'est pas forcément colinéaire à

l'axe du bateau. Le vent réel ainsi obtenu correspond donc bien au vent observé par un utilisateur immobile par rapport au fond et non par rapport à la surface. On parle également de vent-fond.

Dans le monde de la voile, le vecteur vitesse du vent est usuellement décrit sous forme de coordonnées polaires : la vitesse (on parle également de force) correspond à la norme du vecteur, tandis que l'argument est donné sous la forme d'un angle, soit par rapport à l'axe du bateau (ou parle alors d'angle au vent), soit par rapport au nord géographique (on parle alors de direction du vent).

Le vent réel est décrit par sa vitesse TWS (*True Wind Speed*), et soit par sa direction TWD (*True Wind Direction*), soit par l'angle qu'il forme avec l'axe du bateau TWA (*True Wind Angle*). Le vent apparent, directement mesuré par les instruments du bateau, se décrit par sa vitesse AWS (*Apparent Wind Speed*) et son angle AWA (*Apparent Wind Angle*).

Les différents types de vent

Le vent réel est souvent variable, à la fois dans le temps et dans l'espace. Il peut être décrit comme un champ de vecteur tridimensionnel évoluant temporellement. Dans le domaine de la course au large, on fait le plus souvent l'hypothèse que le vent est constant en altitude et peut ainsi être décrit par un champ vectoriel en deux dimensions. Cela revient à négliger les phénomènes de cisaillement du vent (les couches basses, freinées au contact de la surface, sont moins rapides), et le vent est généralement exprimé par les marins à 10 mètres d'altitude [Ber04].

Divers termes sont employés pour décrire la vitesse du vent. L'échelle la plus répandue est l'échelle de Beaufort, qui est présentée tableau 2.1. Elle est composée de 13 degrés, chacun adjoint d'un terme descriptif [Ber04]. Le tableau présente également une correspondance avec les concepts manipulés par les marins décrits par G. Guillou [Gui10]. Notons que les vitesses sont exprimées en nœuds (abrégé *kts* pour *knots*), comme de coutume dans le milieu maritime ($1kts \simeq 0.514m/s$).

Concept	**Beaufort**	**Vitesse** (kts)
Pétole	0	< 1
	1	1 à 3
Petit temps	2	4 à 6
	3	7 à 10
Medium	4	11 à 16
	5	17 à 21
Brise	6	22 à 27
	7	28 à 33
Gros temps	8	34 à 40
	9	41 à 47
	10	48 à 55
	11	56 à 63
	12	> 63

TABLE 2.1 – Échelle de *Beaufort* et concepts associés

Phénomène	**Concept**	**Caractérisation**
Global	Stable	TWS et TWD stables
	Oscillant	TWD et/ou TWS oscillants
	Basculant	TWD tourne
	Oscillant basculant	TWD tourne et oscille
	Forcissant	TWS croissante
	Mollissant	TWS décroissante
Ponctuel	Rafale	Montée importante et brutale de la TWS
	Molle	Chute importante et brutale de la TWS
	Ado	Rotation brutale de la TWD, à gauche si bâbord amure ou a droite sinon
	Refus	Rotation brutale de la TWD, à droite si bâbord amure ou à gauche sinon

TABLE 2.2 – Les variations temporelles du vent

Si les variations spatiales sont très dépendantes du site et des phénomènes météorologiques locaux, les variations temporelles suivent souvent des schémas bien identifiés. Un vocabulaire spécifique est utilisé pour caractériser ces variations. Comme on peut le voir dans le tableau 2.2, les phénomènes se décomposent en deux catégories : les évolutions du vent global et les phénomènes ponctuels et locaux [Gui10].

Les concepts présentés ici affectent soit la direction, soit la vitesse du vent. Cependant, il n'est pas rare que ces phénomènes se combinent pour donner une variation simultanée des deux grandeurs, particulièrement pour les phénomènes locaux (on dit par exemple qu'il y a de l'ado dans la risée, ou du refus dans la molle...).

2.2.2 La mer et son état

L'état de mer, qui est caractérisé par les phénomènes visibles sur la surface, est une composition d'éléments ponctuels (vague isolée) et de phénomènes étendus (courants, houle, trains de vagues). L'étude et la description de l'état de mer est un travail complexe qui relève de l'océanographie. Cette partie n'a pas pour ambition d'entreprendre un tel travail, mais présente simplement les divers concepts utilisés par les marins pour caractériser la mer sur laquelle évolue leur voilier.

La mer du vent correspond aux phénomènes ondulatoires qui apparaissent en surface sous l'effet du vent. Cela peut aller de simples ridules pour les vents faibles à des vagues de plusieurs mètres d'amplitude pour les vents forts soufflant sur de longues distances. La mer du vent est également dépendante des phénomènes de courant. En particulier, un vent de direction opposée au courant lèvera une mer très courte et souvent désordonnée, nommée clapot. Notons que l'observation de la mer du vent est un excellent moyen d'estimer visuellement la force du vent sans instruments.

A la mer du vent, il faut ajouter la houle. Il s'agit d'un phénomène créé par un vent éloigné de la zone d'observation, qui s'est ensuite propagé. La houle, souvent très régulière, est caractérisée par une amplitude (hauteur des vagues), une période (temps séparant deux crêtes) et une longueur d'onde (distance séparant deux crêtes) [Par04].

Lors de phénomènes périodiques, l'amplitude des vagues n'est jamais constante. Pour donner une description significative du phénomène, il est courant de donner la hauteur moyenne du tiers des plus grandes vagues observées, on parle alors de $H_{1/3}$. De manière identique au vent, une échelle est mise en place pour qualifier l'état de mer en fonction de la hauteur des vagues. Il s'agit de l'échelle de *Douglas*, visible tableau 2.3 [Ber04]. Les phénomènes de houle et de mer de vent se combinent pour former un état de mer complexe. Lorsque ces phénomènes possèdent des directions différentes et des amplitudes comparables, on obtient une mer dite *croisée*, qui complique souvent la navigation.

Degrés	**Descriptif**	**Hauteur (m)**
0	calme	0
1	ridée	0 à 0.1
2	belle	0.1 à 0.5
3	peu agitée	0.5 à 1.25
4	agitée	1.25 à 2.5
5	forte	2.5 à 4
6	très forte	4 à 6
7	grosse	6 à 9
8	très grosse	9 à 14
9	énorme	> 14

TABLE 2.3 – Échelle de *Douglas* permettant de caractériser la mer selon la hauteur des vagues

2.2.3 Modélisation de l'environnement

La simulation d'un voilier en interaction avec son environnement nécessite une modélisation des différents éléments qui constituent cet environnement. Pour la modélisation du vent, plusieurs approches sont envisageables, les plus simples étant l'utilisation de données réelles issues de capteurs ou de prévisions météorologiques que l'on peut obtenir des divers instituts météo sous forme

de champs vectoriels [Ber04]. Il est également possible de recréer un vent dont les propriétés statistiques sont semblables à la réalité : on utilise pour cela des approches fréquentielles, souvent avec une modélisation supplémentaire des phénomènes turbulents [Nic+02]. Les modèles ainsi développés sont très dépendants des applications : il est par exemple possible d'avoir des modèles complexes de rafales en trois dimensions [Ric13] qui sont spécialisés pour les études aéronautiques, mais qui s'adaptent difficilement aux problématiques des voiliers. Il faut donc s'assurer de la cohérence des données obtenues avec l'application qui en est faite.

La simulation de l'état de mer est également possible. Au-delà des modèles simples qui considèrent une houle constante en période, amplitude et direction, on trouve des modèles plus évolués basés sur la combinaison et les interactions des différents phénomènes océanographiques [Par04] et le couplage entre la variabilité du vent et l'état de mer [BW00]. Des modèles météorologiques peuvent également fournir des données prévisionnelles, sous forme d'un champ vectoriel représentant l'amplitude et la direction de la houle, doublé d'un champ scalaire pour la période. On obtient des cartes d'état de mer, qui peuvent être fournies sous forme numérique [Ber04].

Cependant, l'interaction entre la mer et les coques et appendices du voilier est un phénomène complexe, beaucoup plus difficile à simuler que l'action du vent sur les voiles. Des approches simplifiées sont possibles en considérant la résultante des efforts hydrostatiques en différents points de mesure [Gui10], mais une approche réaliste du phénomène nécessite d'avoir recours à la simulation numérique pour étudier les écoulements (stationnaires ou non) qui s'établissent autour du bateau. Cela nécessite un important développement, et les calculs considérés sont parfois extrêmement lourds si on y intègre les problématiques de surface libre ou d'interaction fluide/structure [Lom+12]. Ces approches sont souvent utilisées par les architectes navals, dans le but de réaliser des programmes de prédiction de performance (VPP, *Velocity Prediction Program*), mais sont difficilement accessibles aux utilisateurs des bateaux (notons que l'établissement d'une matrice hydrodynamique pour un VPP peut requérir plusieurs semaines de calcul sur un cluster dédié).

Comme les modèles d'interaction entre le voilier et l'état de mer sont une problématique délicate, la suite de ce travail suppose que, pour une mer *calme* à *belle*, cette interaction a peu d'effet sur le comportement du voilier et sa manière de le piloter. La simulation de l'environnent se réduit donc à la simulation du vent sur le plan d'eau. Cependant, le travail d'intégration des modèles de vent qui est proposé au chapitre 8 est transposable à l'état de mer, pour peu que l'on dispose d'un modèle d'interaction entre la mer et le voilier.

2.3 L'électronique embarquée sur un voilier de compétition

Les voiliers de course actuels sont des prototypes fortement instrumentés. Ils disposent d'une électronique embarquée dont l'objectif est triple : fournir des éléments de performance et de sécurité en temps réel au skipper, fournir une représentation fidèle de l'environnement au pilote automatique et enregistrer l'ensemble des données du bateau pour des analyses post-navigation, tant pour l'analyse des performances que pour la traçabilité en cas de problème technique ou de casse matérielle.

2.3.1 Les capteurs

Les capteurs fournissent des données sur l'environnement immédiat du bateau ainsi que sur son état interne. D'un bateau à l'autre, le nombre et le type de capteurs peuvent varier fortement. Cependant, on retrouve une partie commune indispensable à la navigation permettant de mesurer le vent apparent, ainsi que la position, l'attitude et la vitesse du bateau. Ces incontournables sont listés tableau 2.4, ainsi que la nature des données qu'ils produisent.

Cette liste n'est pas exhaustive, et l'on trouve de nombreux autres capteurs sur la plupart des grands voiliers de course, que ce soit pour compléter les données environnementales (capteur de température d'eau ou d'air, baromètre), l'attitude du bateau (capteur de hauteur par rapport à la surface de l'eau pour les bateaux volants), les efforts dans les différentes structures (jauges de contrainte, mesure de déformation par fibres optiques, capteurs de pression pour les actionneurs hydrauliques) ou encore l'état du bateau (position des appendices, des voiles, état de remplissage des ballasts, détection d'eau dans les coques...).

Capteur	Type de données	Nombre de données	Nature des données
Aérien	Vent apparent	2	AWS
			AWA
GPS	Position	4	Latitude et longitude
			COG et SOG
Compas	Cap	1	Cap
Speedomètre	Vitesse	1	SOW
Centrale inertielle	Attitude	9	3 accélérations
			3 angles d'Euler
			3 vitesses angulaires

TABLE 2.4 – Les principaux capteurs du voilier

2.3.2 Les actionneurs

Si les capteurs sont nombreux et variés à bord des engins de course, il n'en va pas de même pour les actionneurs. En effet, dans les Règles de Course à la Voile, la RCV n ° 52 précise que « le gréement dormant d'un bateau, son gréement courant, ses espars et appendices mobiles de coque doivent être réglés et manœuvrés uniquement par la force fournie par l'équipage » [Féd17], ce qui interdit tout asservissement.

Dans le cadre de la course au large, et en particulier pour les navigations solitaires, une dérogation est faite à la RCV 52 pour autoriser l'asservissement du ou des safrans autour de leur axe principal (axe de lacet), ce qui permet la mise en place d'un pilote automatique. Les seuls actionneurs présents sur les voiliers de course servent donc à manœuvrer les safrans. Il s'agit soit de vérins électriques en prise sur un palonnier relié à la mèche du safran, soit d'un moteur électrique entraînant une chaîne reliée aux drosses de barre. On trouve également des vérins hydrauliques actionnés par une pompe électrique. Dans tous les cas, la puissance est tirée d'un parc de batteries, qui sont chargées par divers moyens (panneaux solaires, éolienne, hydrolienne, génératrice diesel...)

2.3.3 L'architecture matérielle

L'architecture matérielle d'une installation électronique embarquée respecte le plus souvent le schéma standard qui est présenté figure 2.5. Les différents capteurs sont reliés à des cartes d'acquisition, qui transmettent les grandeurs mesurées à une centrale de navigation. Cette centrale, qui fait office de calculateur principal, recueille et traite l'ensemble des données mesurées, dans le but de reconstruire une vision de l'environnement du bateau et de son attitude. La centrale est, entre autres choses, chargée d'évaluer le vent réel.

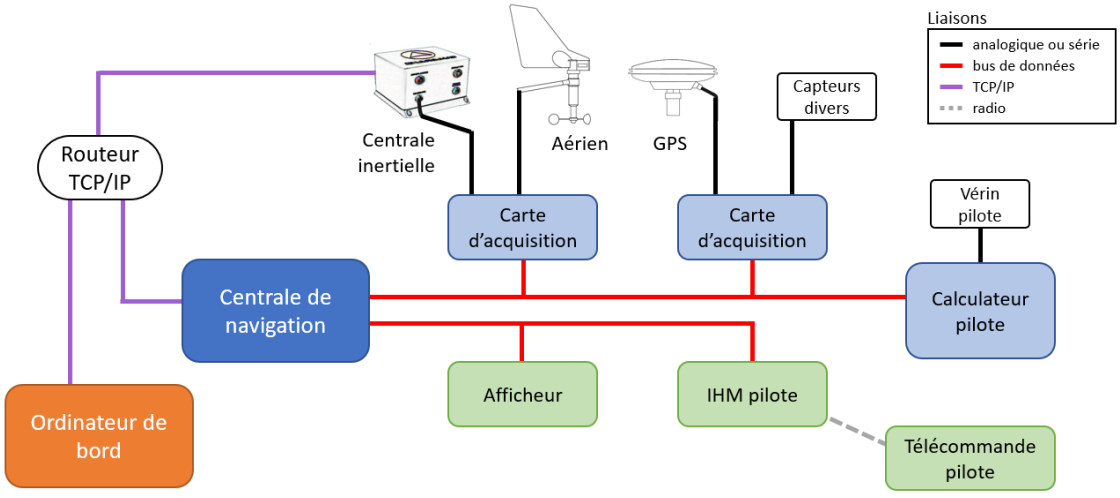

FIGURE 2.5 – Description générique de l'architecture matérielle embarquée sur un voiler de course

Le calculateur du pilote automatique est souvent physiquement dissocié de la centrale de navigation. Il reçoit de cette dernière toutes les informations nécessaires au pilotage, et il fournit une consigne à l'actionneur qui agit sur la barre.

Un ordinateur de bord est utilisé par le skipper pour effectuer les analyses météo et élaborer la stratégie à suivre (routage), ainsi que pour gérer la navigation (cartographie) et les communications par satellites. Il est souvent interfacé avec la centrale de navigation à des fins de configuration et d'affichage.

Les liaisons entre la centrale de communication, les cartes d'acquisition et les afficheurs sont réalisées via des bus de communications, souvent basés sur la technologie des bus de données CAN (*Controller Area Network*). Il arrive que les cartes soient embarquées directement dans la centrale. La connexion entre la centrale de navigation et l'ordinateur de bord se fait classiquement par réseau IP. Ce même réseau peut servir à la configuration de capteurs complexes, telles les centrales inertielles.

2.3.4 Les pilotes automatiques actuels

Les pilotes automatiques des voiliers de compétition sont des calculateurs qui délivrent une unique commande à l'actionneur du safran. Ils ne contrôlent directement qu'un degré de liberté du bateau, la rotation en lacet. Le calculateur reçoit l'ensemble des données fournies par la centrale de navigation, sous forme d'un flux de données numériques (exprimées en virgule flottante, assimilable au type *double*), à des fréquences variant généralement entre 1 Hz et 100 Hz.

Il existe plusieurs marques de pilote, fournies par les fabricants d'électronique marine. La plupart des équipes de course au large utilisent les mêmes modèles, issus de trois marques. Bien qu'ils présentent des différences notables, leur mode de fonctionnement général est identique :

- Plusieurs modes de régulation sont disponibles : soit autour d'un cap, soit autour d'un angle au vent (apparent ou réel). La consigne est réglée à la valeur courante de la donnée au moment où le pilote est enclenché, et peut ensuite être modifiée manuellement.
- La loi de contrôle ou la consigne peuvent être modifiées temporairement au sein du pilote pour intégrer des éléments de performance ou de sécurité.
- La loi de contrôle n'est pas accessible directement à l'utilisateur.
- Des paramètres peuvent être réglés pour adapter la loi de contrôle au bateau et aux conditions rencontrées. Ces paramètres peuvent être nombreux (parfois plusieurs centaines), et la mise au point est souvent délicate, d'autant que la signification exacte et les actions de certains paramètres ne sont pas toujours explicites...

Les pilotes peuvent être vus comme des boîtes grises dans lesquelles l'utilisateur ne maîtrise que le paramétrage de la loi de contrôle. La mise au point d'un pilote est souvent effectuée avec le fabricant ou le revendeur de la solution. Les interfaces homme/machine permettant de contrôler le pilote (définition de la consigne, du mode, réglage des paramètres de contrôle) sont variables selon les modèles et les marques des pilotes. Elles peuvent être confiées à des afficheurs dédiés munis de boutons, ou bien utiliser les afficheurs de la centrale de navigation, ou encore passer par l'ordinateur de bord. Dans les configurations solitaires, une télécommande sans fil permet souvent de prendre la main sur le pilote automatique.

2.4 Discussion sur le pilotage des voiliers de compétition

L'objectif de ce travail est de proposer des moyens logiciels permettant la mise au point et le paramétrage d'un pilote ou d'une loi de contrôle via la simulation. Nous avons vu que l'architecture matérielle embarquée sur les voiliers de compétition est déjà déterminée : articulée autour d'un paradigme de flux de données, la centrale fournit des informations au calculateur du pilote qui n'agit que sur la direction. La simulation des lois de contrôle devra rester compatible avec l'architecture matérielle existante.

La simulation implique de disposer d'un environnement logiciel pour exécuter la loi de contrôle, mais également de modèles pour reproduire le comportement du bateau et de l'environnement dans lequel il évolue. Divers modèles sont disponibles pour ces deux éléments, avec des complexités et des spécificités variées. En fonction des objectifs de la simulation, certains modèles simples peuvent

être suffisants. Il faut de plus garder une cohérence entre les modèles de simulation utilisés : simuler l'état de mer très précisément n'aura que peu d'intérêt si le modèle de voilier utilisé ne prend pas cet état en compte. On comprend ici l'intérêt d'un simulateur modulaire, où il est possible de choisir les modèles utilisés lors de simulations.

Le voilier, l'environnement dans lequel il évolue et le contrôleur embarqué constituent des entités indépendantes, issues de domaines différents, et interagissant fortement entre elles. C'est une problématique commune à un grand nombre de systèmes cyber-physiques [BG11] : chaque domaine dispose de ses conventions, de son vocabulaire, de moyens dédiés en termes de modélisation et de simulation. Il est donc nécessaire de faciliter l'intégration d'éléments hétérogènes afin de modéliser les interactions entre ces éléments.

L'intégration passe par la mise en place d'une architecture cohérente entre les composants, la définition d'interfaces et la standardisation des échanges entre les constituants de cette architecture. Les chapitres qui complètent cette partie présentent les méthodes applicables pour la mise en place d'une telle architecture.

Chapitre 3

L'Ingénierie Dirigée par les Modèles

Pour construire un simulateur, il est toujours possible de le coder directement dans un langage de programmation tel que C++ ou Matlab. Une telle approche, centrée sur le code, est bien sûr fortement liée à une technologie et à un langage, de programmation. La maintenance, la validation et l'évolution de l'application sont difficiles car l'approche centrée sur le code pose de nombreux problèmes :

- comment réutiliser du code existant, parfois écrit dans un autre langage ?
- comment intégrer des briques logicielles hétérogènes ?
- comment faire évoluer le simulateur pour de nouveaux besoins ?
- comment tester chaque brique du simulateur ?

Pour résoudre ces problèmes classiques d'ingénierie logicielle, l'Ingénierie Dirigée par les Modèles (IDM) [Sch06 ; BCM12 ; Da 15] propose une approche non pas centrée sur le code, mais basée sur un modèle du système que l'on veut développer.

Le principe de l'IDM est de résoudre des problèmes d'ingénierie logicielle classiques en se focalisant sur la modélisation et en fournissant des modèles et des outils pour appréhender le système, ses besoins et ses spécificités. Dans notre cas, le problème d'ingénierie logicielle est de construire un simulateur du comportement d'un voilier, qui est un système cyber-physique évoluant dans un environnement complexe. Selon une approche IDM, construire un simulateur implique la création de modèles du système cyber-physique, de son environnement, mais également du simulateur lui-même. L'objectif de cette partie est d'étudier les possibilités qu'offre l'IDM pour le développement d'un logiciel de simulation d'un système cyber-physique, et les enjeux autour d'une telle modélisation.

3.1 La modélisation

Comme son nom l'indique, l'IDM consiste à mettre les modèles au cœur du processus de développement. Pour cela, il est important de bien cerner la notion de modèle, et d'être conscient des possibilités apportées par le processus de modélisation.

3.1.1 Les principes de modélisation

Un modèle est une représentation de certains aspects d'un système et de son environnement [Völ+13]. Il s'agit d'une abstraction du système, qui permet d'exprimer aussi bien ses aspects fonctionnels que ses aspects extra-fonctionnels. L'objectif est de reporter les préoccupations non plus sur le logiciel, mais sur les caractéristiques du système et des problématiques qui lui sont spécifiques [Sch06]. Ainsi, le modèle d'une application est indépendant du langage de programmation utilisé pour la développer et de la plateforme sur laquelle elle est déployée.

La modélisation sert à cerner, via différentes vues, plusieurs aspects du même système : sa structure, son comportement, les données qui y transitent, les contraintes qui s'y appliquent. On fait donc une distinction entre le modèle au sens de l'IDM et les technologies qui permettent de programmer une application via la modélisation d'un aspect, généralement le comportement,

telles que les langages *Modelica* [MEO98] et *Simulink* [Mat18]. Ces derniers ne permettent pas d'appréhender tous les aspects du système à travers de modèles dédiés.

Dans le domaine visé, la modélisation s'appuie généralement sur un diagramme de classe représentant les concepts à manipuler ainsi que les liens qui les relient [OMG17b ; Ecl19b].

3.1.2 Les bénéfices de l'IDM

L'IDM est souvent utilisée lors de la phase de conception d'un projet, afin d'apporter une réponse efficace dans divers domaines :

La conception de l'architecture : l'IDM permet au concepteur d'un système de se concentrer sur l'architecture de celui-ci, sans se soucier de contraintes d'implémentation liées à une technologie particulière. Le modèle permet de décrire les spécifications et les besoins du système d'un point de vue fonctionnel et extra-fonctionnel.

La vérification : des outils de vérification de modèles peuvent être utilisés. La vérification passe souvent par l'expression de contraintes sur le modèle. Des outils de vérification permettent ensuite de s'assurer que les contraintes sont bien respectées. L'expression de ces contraintes peut être intégrée au modèle ou se faire via un langage dédié, tel que l'*Object Constraint Language* (OCL [OMG14]).

La documentation : les différentes vues du modèle permettent d'aborder le problème sous ses divers aspects, et ainsi de faciliter la compréhension de celui-ci.

L'automatisation : le fait de disposer de moteurs de transformation et de générateurs de code permet d'obtenir une application pleinement conforme aux règles de conception imposées dans le modèle. On parle d'approche *correct by construction*, qui est à opposer à l'approche traditionnelle *construct by correction* [Sch06].

3.1.3 Les niveaux de modélisation

Les modèles sont constitués d'objets de types définis. Avant de pouvoir créer et éditer ces modèles, il est nécessaire de définir l'ensemble des types utilisables et leurs relations. Cette définition s'apparente à la création d'un typage de modèles, que l'on nomme méta-modèle. La définition d'un méta-modèle permet de préciser la description du domaine. Le modèle peut ensuite être traité par les experts du domaine et non simplement par les experts de la programmation [Sch06 ; FR07 ; BCM12].

Afin de définir les éléments du méta-modèle et d'éviter une récursivité infinie sur les niveaux d'abstraction, l'Objet Management Group (OMG) propose un standard, le *Méta Object Facility* (MOF [OMG16]), qui comprend quatre couches successives, chaque couche venant instancier celle du dessus :

1. Le méta-méta-modèle MOF, qui s'auto-décrit, permettant de construire des méta-modèles
2. Les méta-modèles, permettant de construire des modèles
3. Les modèles, permettant d'appréhender les systèmes
4. Les systèmes (le monde réel, l'application)

Plusieurs outils permettent de créer et d'éditer des méta-modèles. On peut notamment citer le framework EMF d'Eclipse [Ecl19b], dont le méta-méta-modèle *Ecore* est proche de la spécification MOF.

3.2 Les outils autour de l'IDM

L'IDM place le modèle au centre du processus de développement. Cependant, pour profiter pleinement des apports de l'approche, il faut disposer d'outils qui permettent de manipuler efficacement les modèles et de leur appliquer divers traitements afin d'en extraire des informations. Ainsi, les frameworks qui permettent la manipulation de modèles pour la mise en place de l'IDM proposent également un outillage autour de ces modèles.

3.2.1 Les éditeurs de modèles

Les éditeurs permettent de créer et de modifier les modèles. Cela passe par l'instanciation et la manipulation des objets présents dans le méta-modèle. Selon les besoins, il est fréquent d'avoir un ou plusieurs éditeurs pour chaque méta-modèle. Plusieurs éditeurs permettent de décrire des aspects différents du même modèle, on parle alors de différents points de vue.

Un éditeur de modèle se base sur une syntaxe concrète, qui permet de faire le lien entre les éléments présents dans l'éditeur et le modèle. Les frameworks actuels proposent deux principaux types de syntaxes : les syntaxes textuelles, et les syntaxes graphiques [BCM12]. Il en résulte deux catégories d'éditeurs :

Les éditeurs textuels permettent d'instancier les modèles au travers de langages, souvent de forme déclarative. Les éditeurs peuvent proposer des fonctions plus ou moins évoluées, telles que la vérification syntaxique, la coloration syntaxique, la mise en forme automatique ou encore l'auto-complétion. Plusieurs projets permettent de définir une syntaxe textuelle pour un méta-modèle donné, permettant ainsi la création d'un éditeur dédié. Les plus communs sont *Xtext* [Ecl19e] ou encore *EMFText* [Hei+11]

Les éditeurs graphiques proposent, comme leur nom l'indique, d'instancier les modèles via des vues graphiques. Ces vues sont souvent sous forme de diagrammes, d'arbres ou de tables. Plusieurs vues du même modèle sont possibles. Les éditeurs graphiques sont particulièrement utilisés pour une visualisation synthétique de la structure du programme, et la modélisation des flux et des connexions entre les éléments du modèle [LT12]. Les éditeurs graphiques sont outillés pour permettre la modification du modèle. La mise à disposition d'une palette d'outils permet de créer des objets et de les lier entre eux depuis la vue graphique. Plusieurs projets permettent de créer des éditeurs graphiques à partir d'un méta-modèle, via la définition d'une syntaxe graphique, comme le *Graphical Modeling Project* [Ecl19c] avec les outils GMF et Graphiti, ou encore le projet *Sirius* [Ecl19d].

Le choix entre une syntaxe textuelle ou graphique dépend de la nature du système modélisé et des objectifs de la modélisation, mais également des préférences personnelles de l'utilisateur de l'éditeur. Les syntaxes graphiques permettent généralement une meilleure compréhension du modèle par les experts du domaine visé et facilitent les échanges entre les divers intervenants, tandis que les syntaxes textuelles sont souvent plus brèves. Cependant, les deux approches ne sont pas forcément exclusives : certains projets combinent les deux, offrant ainsi divers points de vue sur les modèles [Add+17].

3.2.2 Le besoin d'intégration

La modélisation proposée par l'IDM n'est pas une finalité, c'est un outil pour faciliter le développement. Pour maximiser leur utilité, il est important que les modèles puissent s'intégrer avec d'autres outils. Usuellement, les frameworks proposant de l'IDM embarquent des outils pour réaliser diverses actions sur le modèle ou à partir de lui [Sch06 ; BCM12]. On peut citer entre autres la vérification de modèles, la gestion de lignes de produit, l'intégration avec des intergiciels, la transformation de modèles et la génération de code.

L'IDM permet de faire le lien entre ces différents outils, d'analyser et de transformer les informations relatives à la structure et au comportement, les données et les propriétés extra-fonctionnelles de l'application. Dans notre étude, à partir de modèles de simulation, l'objectif principal est de générer le code du simulateur. La partie suivante détaille donc l'outillage de l'IDM autour de la génération de code.

3.2.3 La génération de code

L'exploitation du modèle pour le développement de logiciel peut se faire de deux manières différentes : la génération de code ou l'interprétation du modèle. Dans la seconde option, un moteur générique (interpréteur) est implémenté, ce qui permet d'analyser et d'exécuter le modèle *à la volée*, à l'image des langages interprétés. Si cette approche apporte de la flexibilité en permettant la modification du modèle à l'exécution, elle nécessite l'installation de l'interpréteur sur la plateforme cible et n'égale pas les performances du code généré [BCM12]. Pour le développement d'un simulateur, la génération de code semble plus adaptée. Cette partie présente les différentes options qui se présentent lors de la mise en place d'un générateur de code.

Le langage et la plateforme

La génération de code permet de s'abstraire d'une technologie particulière lors de la conception du modèle. Le générateur se charge lui-même des adaptations nécessaires pour la technologie choisie. Le principal avantage de la génération est la conformité de l'application avec le modèle, le passage de l'un à l'autre étant entièrement automatisé.

La génération permet la réutilisation d'un même modèle pour générer des programmes adaptés aux spécificités de différentes plateformes, potentiellement dans des langages différents [GB16]. Il est également possible de générer du code à destination d'un intergiciel (*middleware*) ou d'un framework spécifique afin de circonscrire le code généré au domaine modélisé.

Le type de générateur

Il est possible de générer du code de plusieurs manières différentes à partir d'un modèle. On distingue deux principaux types de générateurs [BCM12] : les générateurs axés sur un langage de programmation, et les générateurs basés sur des templates.

Les générateurs basés sur un langage de programmation génèrent, dans un langage donné, du code permettant de créer une arborescence d'objets à l'image du contenu du modèle. De tels générateurs accompagnent souvent les éditeurs de modèles : ils offrent une API d'accès aux objets du modèle dans le langage cible, ce qui permet à des outils externes de manipuler le modèle. Le fonctionnement du framework EMF [Ecl19b] repose en partie sur un générateur de ce type : une API Java est générée pour les modèles basés sur *Ecore*, ce qui permet à de multiples outils de manipuler ces modèles. En général, ce type de générateur n'est pas paramétrable, le code produit dépend uniquement du contenu du modèle.

Les générateurs basés sur des templates [SLS18] mettent en œuvre une transformation du modèle vers du texte, qui a été standardisée par l'OMG sous le nom de *MOF Model to Text Transformation* (MOFM2T) [OMG08]. Dans cette approche, indépendante d'un langage de programmation particulier, le code généré est décomposé en deux parties : une partie statique qui ne dépend pas du contenu du modèle, et une partie dynamique. La génération se fait à partir d'un template, qui contient les parties statiques ainsi que les instructions permettant d'extraire du modèle et de mettre en forme les parties dynamiques. Le template prend l'allure d'un code à trous, ces derniers étant remplis par la partie dynamique. Des mécanismes de requêtes dans le modèle et des structures de contrôle (itérations et conditions) permettent de faire correspondre le code généré au contenu du modèle.

La génération basée sur des templates permet de générer du code dans des langages variés, ce qui assure une bonne adéquation avec la plateforme et l'application cible. Le résultat de la génération est une combinaison des informations contenues dans le modèle et dans les templates du générateur, ce qui offre des possibilités de personnalisation importante du code produit (un même générateur peut produire du code adapté à différentes plateformes à partir du même modèle, en utilisant des templates différents). Cependant, cette technologie implique la création de templates adaptés pour chaque méta-modèle. De nombreux outils de génération de code basés sur des templates sont disponible pour divers usages : Xpand, XSLT, JET, MOFScript... L'un des plus populaires est le projet *Acceleo* [Ecl19a], une implémentation du standard MOFM2T qui fonctionne avec les modèles EMF.

Génération totale ou partielle

La génération de code consiste à produire des fichiers textuels contenant du code dans un langage de programmation donné, à partir d'informations contenues dans un modèle. Ces fichiers sont par la suite intégrés dans un environnement de développement adapté au langage cible, afin d'être compilés puis exécutés au même titre que le code édité manuellement.

En fonction de la richesse des modèles utilisés et de la complexité de l'application à produire, il est possible de générer l'ensemble (on parle alors de génération totale) ou seulement une partie du code (génération partielle). Dans cette seconde option, le code généré devra être complété par le développeur. Il est alors nécessaire de prévoir des mécanismes de protection pour éviter que les modifications apportées ne soient écrasées par une nouvelle phase de génération [.]

La génération totale permet de produire de manière automatique un logiciel exécutable. Pour

cela, il est nécessaire que toutes les informations intervenant dans le développement d'un tel logiciel soient contenues dans le modèle, ce qui implique souvent un modèle très riche et donc lourd à mettre en place. La génération partielle offre plus de souplesse pour un simulateur. Seules les informations pertinentes pour la composition de modules et la simulation sont modélisées. Les codes métier spécifiques restent du domaine du spécialiste. Cette approche facilite grandement l'intégration de code existant.

Intégration de code existant

Dans le cas de la génération partielle, il est nécessaire de compléter le code généré avant de pouvoir l'exécuter. Cette situation se produit lorsque l'on veut réutiliser du code spécifique existant (on parle alors de code métier, ou *legacy code*). L'intégration du code métier est un des défis de la génération de code, et plusieurs solutions sont possibles pour la réaliser [Gre+15] :

- modéliser le comportement du code métier, pour ensuite le générer. Cette solution évite les problématiques de génération partielle, mais impose une complexification importante (voire impossible) du modèle.
- copier le code métier au sein du code généré. Cette solution est sans doute la plus rapide, et peut être efficace si des mécanismes de protection sont mis en place pour éviter que le code ne soit écrasé par une nouvelle génération. Cependant, le code généré doit être suffisamment spécifique pour s'interfacer avec le code métier (échange de données, appels de fonctions...). D'une manière pragmatique, lors de la mise en place d'une telle solution, il est courant de modifier le code métier pour en adapter l'interface avec le code généré. En fonction du langage de programmation, il est possible d'utiliser une commande d'inclusion afin de séparer le code généré et le code métier dans des fichiers différents.
- utiliser une interface standard : si le code métier a été conçu comme une bibliothèque, il dispose d'une API connue qui permet de l'utiliser. Le code généré peut donc réaliser des appels sur cette API pour effectuer des actions spécifiques. On parle alors de délégation : le programme généré délègue une partie du traitement au code métier. C'est une approche utilisée communément dans les processus de réutilisation de code [Gam+95] car elle permet une indépendance des deux parties du code, qui interagissent uniquement à travers l'API.
- étendre le code généré par héritage dans le code métier. Cette approche est utilisée lorsque le code généré propose des interfaces et des implémentations de base, et que le code métier vient compléter ou modifier ces implémentations par héritage. Cependant, comme il est nécessaire de connaître les interfaces générées lors de l'écriture du code métier, cette méthode ne permet pas à proprement parler d'intégrer du code déjà existant.
- étendre le code métier par héritage dans le code généré. Cette approche est sensiblement similaire à la délégation, mais elle nécessite que le code métier et le code généré soient tous deux dans le même langage orienté objet. Le code métier ne fournit pas une API à proprement parler, mais un ensemble de classes respectant un certain standard. Le code généré vient hériter de ces classes, et peut ainsi accéder au comportement du code métier, les échanges étant facilités (attributs de classes partagés, possibilité de surcharger certains comportements) par rapport à une communication à travers une API.

Ces méthodes ne sont pas forcément exclusives, et leur utilisation peut varier en fonction de la quantité et de la nature du code métier à intégrer. La liste n'est pas exhaustive, d'autres méthodes d'intégration existent, souvent basées sur des outils tiers permettant la fusion de code, ou sur des spécificités de certains langages, comme les classes partielles [Gre+15]. Pour conserver les avantages de la génération de code, il est cependant important que la méthode d'intégration choisie requière un minimum d'effort. Les développeurs peuvent ainsi se consacrer entièrement au développement du code métier, qui représente la valeur ajoutée du programme [WHR14].

3.3 Approche générique ou spécifique

Deux approches concurrentes sont possibles lorsque l'on souhaite faire de la modélisation : une approche à base de modèles généralistes, et une approche à base de modèles spécifiques. Si on reprend la hiérarchie MOF présentée ci-dessus, la première approche revient à utiliser un méta-modèle (deuxième niveau) générique existant tels que UML [OMG17b] ou SysML [OMG17a], la seconde revient à définir son propre méta-modèle, spécialisé pour le domaine d'application visé, ici la simulation de lois de contrôle pour un voilier.

3.3.1 Les langages génériques

Les langages de modélisation génériques (GPML pour *General Purpose Modeling Language*) sont des langages permettant de modéliser des systèmes de manière générique, applicables à tous les domaines, indépendamment de tout contexte. Ils proposent des structures permettant la représentation générique des concepts utilisés en modélisation. Plusieurs GPML sont disponibles pour modéliser les systèmes, chacun ayant des spécificités dans son approche. Les plus répandus sont :

UML (*Unified Modeling Language*) : il s'agit d'un langage de modélisation générique standardisé par l'OMG [OMG17b]. UML propose plusieurs vues sur le système modélisé, au travers de multiples diagrammes (14 en UML 2.3). Les divers modèles permettent de documenter tous les aspects du processus de développement. Le langage a été largement adopté pour le développent logiciel [Val11]. Il est également possible d'utiliser UML pour modéliser d'autres concepts que le développement logiciel, en le couplant avec des outils tels que *Simulink* [Mro02] ou *Modelica* [Mro03].

SysML (*System Modeling Language*) : il s'agit d'une extension d'UML adaptée à la modélisation système [OMG17a]. De nouveaux diagrammes sont ajoutés pour exprimer la structure et le comportement, les besoins et la paramétrisation des modèles [CSC08]. L'intégration avec d'autres modèles de simulation et d'analyse est facilitée [Thr10].

AADL (*Architecture Analysis & Design Language*) : il s'agit d'un langage textuel et graphique, capable de modéliser la structure, le comportement et la plateforme d'exécution de systèmes. Il est spécialement conçu et adapté pour décrire les architectures des systèmes temps réel [Fei+04 ; Las+09].

3.3.2 Les langages spécifiques

L'utilisation d'un méta-modèle spécifique au domaine d'application du système permet la manipulation de concepts spécifiques. Pour cela, on a recours à un langage de modélisation spécifique (DSML pour *Domain Specific Modeling Language*). On peut définir le DSML comme un langage bref, souvent déclaratif, qui permet de traiter les éléments d'un domaine particulier [VKV00]. Le DSML est lié à un méta-modèle, il permet de décrire des modèles conformes à ce dernier.

Comme le DSML est très proche du domaine d'application, il peut être facilement compris et manipulé par les experts du domaine. Il présente également l'avantage de pouvoir être en permanente évolution, afin de suivre les évolutions du domaine modélisé [Tha+17]. Cependant, son utilisation nécessite la construction d'un éditeur dédié, basé sur une syntaxe concrète faisant le lien entre le DSML et les éléments du méta-modèle. Il requiert également la mise en place d'un outillage spécifique pour la génération de code.

3.3.3 Discussion sur les approches spécifiques et génériques

L'approche générique présente l'avantage de proposer des langages unifiés pour les différents domaines. Cela simplifie les échanges d'informations et facilite la compréhension des modèles. Cependant, des mises en œuvre supplémentaires sont nécessaires pour pouvoir adresser les éléments du domaine, certains outils permettent d'étendre les langages génériques pour y amener des spécialisations [FR07]. L'approche la plus courante est la création de profils pour UML [Val11]. Un profil est un ensemble de stéréotypes (classes avec des contraintes), de tags et de contraintes qui permettent la spécialisation des diagrammes UML. Un certain nombre de profils ont été développés pour étendre l'utilisation d'UML en dehors de la stricte programmation orientée objet [CSC08]. Ils adressent des domaines tels la gestion du temps réel [HDB17], les systèmes embarqués ou les systèmes hybrides.

A l'inverse, l'approche spécifique spécialise les langages, permettant une expression des modèles dans un langage proche des conventions du domaine. Cependant, elle demande la création d'un méta-modèle spécifique, ce qui représente du temps de modélisation supplémentaire. Ainsi, la recherche d'efficacité et de pragmatisme amène souvent le concepteur du DSML à s'inspirer de certaines pratiques présentes dans les langages génériques [Tha+17]. Cette approche spécifique offre la possibilité de manipuler simplement et dans une syntaxe « naturelle » les concepts du domaine, elle est donc particulièrement intéressante dans un domaine technique ou les experts ne sont pas forcément informaticiens.

3.4 Discussion sur la modélisation

L'approche proposée par l'IDM facilite le développement et la production d'une application respectant des principes de génie logiciel. Elle permet de s'abstraire d'une technologie particulière en se concentrant sur les spécificités du domaine visé. La génération de code permet ensuite de créer une application dans une technologie adaptée aux besoins du domaine. Comme nous l'avons vu dans ce chapitre, l'utilisation d'une telle approche soulève plusieurs questions :

- en termes de modélisation, il est possible d'utiliser un langage générique (éventuellement de l'étendre pour le spécialiser) ou de développer un langage spécifique ;
- en termes d'outillage, les modèles peuvent être manipulés à travers plusieurs points de vue, à l'aide d'éditeurs textuels ou graphiques ;
- en termes de génération de code, deux grands types de générateurs sont envisageables : ceux axés autour d'un langage de programmation, et ceux basés sur des templates ;
- en termes d'intégration du code métier dans l'application générée, diverses stratégies sont envisageables pour réduire l'effort d'intégration et faciliter la réutilisation du code existant.

Les réponses apportées à ces questions dépendent des objectifs de la modélisation, mais également du domaine d'application. Pour les systèmes cyber-physiques, un point essentiel est la modélisation de l'architecture du système, soit la description des entités modélisées et de leurs interactions. Il faut donc choisir ou établir un modèle permettant de représenter les éléments du système cyber-physique et leur comportement au cours du temps. Le chapitre suivant se concentre ainsi sur l'étude des modèles à composants qui permettent de construire une telle architecture.

Chapitre 4

Modèles à composants pour systèmes cyber-physiques

Les problématiques rencontrées lors de la modélisation d'un système cyber-physique amènent à décomposer celui-ci en plusieurs constituants (logiciels, matériels, mécaniques) reliés entre eux, afin de modéliser les divers composants du système. Si cette approche orientée composant est courante, il existe de nombreux types de modèles à composants. Certains sont très spécifiques et liés à une problématique bien identifiée alors que d'autres sont plus génériques.

Pour notre étude, il est nécessaire de prendre en compte les diverses problématiques liées à la simulation et celles liées à la conception du simulateur lors de la définition d'un système de composants. Dans ce chapitre, nous étudions les caractéristiques des composants et leurs capacités à modéliser un système cyber-physique dans un but de simulation.

4.1 Définitions

Nous commençons par définir le sens donné au terme « composant » dans ce document. La notion de composant varie en fonction du contexte dans lequel elle est utilisée. En ingénierie logicielle, une des définitions les plus communément admises du composant est celle proposée par Clemens Szyperski dans [Szy02] :

> « A software component is a unit of composition with contractually specified interfaces and explicit context dependencies only. A software component can be deployed independently and is subject to third-party composition. »

On voit apparaître ici deux notions fondamentales qui sont la contractualisation des interfaces du composant (entraînant une déclaration des dépendances), et la composition, donc la possibilité d'assembler plusieurs composants pour créer un système. Cette définition reste cependant liée à l'ingénierie logicielle, où les composants désignent des morceaux de code source ou de fichiers exécutables qui peuvent être assemblés et déployés afin de créer une application exécutable.

Dans [Kru04], Philippe Kruchten propose d'élargir la définition de composant pour englober des éléments qui ne sont pas purement logiciels :

> « A component is a nontrivial, nearly independent, and replaceable part of a system that fulfills a clear function in the context of a well-defined architecture. A component conforms to and provides the physical realization of a set of interfaces. »

On y retrouve bien entendu le concept d'interface, mais il apparaît également la notion de fonction réalisée, le composant étant considéré comme une sous-partie cohérente d'un système. Il peut s'agir d'un composant logiciel, mais également d'un système mécanique, d'un modèle mathématique, d'un calculateur indépendant, ou encore de la modélisation d'un de ces éléments. On retient dans ce qui suit qu'un composant peut désigner une partie ou le modèle d'une partie d'un système cyber-physique et de son environnement.

Ces définitions nous amènent à envisager un composant selon deux niveaux [Hoš+10 ; Kru04] :

Les composants de développement interviennent lors de la conception du système (*design-time*). Ils sont principalement utilisés à des fins d'analyse, de vérification, de simulation, de

génération de code et de documentation. En assemblant les composants, le concepteur peut structurer le système étudié pour définir son architecture [LLE07 ; Thr10].

Les composants d'exécution interviennent lors de l'exécution du système (*run-time*). Ils sont principalement utilisés pour leur facilité de déploiement, de réutilisation, ainsi que la possibilité de découvrir et éventuellement de reconfigurer le système lors de l'exécution.

Ces deux approches ne sont pas exclusives. Différentes techniques permettent de transposer les composants de développement en composants d'exécution [NBP13]. En fonction des systèmes, la notion de composant peut également disparaître à l'exécution. Dans la suite de ce travail, seuls les composants de développement sont envisagés. En effet, les problématiques liées aux composants d'exécution sortent du cadre de la modélisation et de la simulation d'un système cyber-physique.

4.2 Apport des composants

La décomposition d'un système en composants est très répandue, tant en modélisation système qu'en génie logiciel. Elle amène de nombreux bénéfices, dont la plupart sont directement exploitables dans le cas de la modélisation d'un système cyber-physique.

La séparation des préoccupations : les différents intervenants sur un système cyber-physique (informaticiens, électroniciens, mécaniciens...) peuvent se concentrer sur leur cœur de métier, qui est encapsulé dans un ou plusieurs composants [HL95 ; BG11].

La réutilisation : la création de bibliothèques de composants spécialisés dans un domaine permet de modéliser simplement le système par l'assemblage d'éléments existants [LT12 ; DTL15], ce qui réduit l'effort nécessaire à la création de nouveaux systèmes.

La facilité d'évolution : si les besoins ou les missions du système évoluent, le découpage en composants facilite son adaptation : seules les parties impactées par les évolutions doivent être modifiées [NBP13].

La gestion de l'hétérogénéité : les systèmes cyber-physiques comportent des constituants hétérogènes par leur nature (capteurs, actionneurs, logiciels, modèles, environnements physiques...) [Rom+10], par la nature des données qu'ils échangent, par leur caractéristique temporelle (synchrone, asynchrone, temps réel ou simulé) [BBS06], mais également par leur granularité. L'approche par composants peut permettre une standardisation des interfaces et la définition d'architectures pour l'intégration de différents éléments et pour la construction de systèmes hétérogènes complexes [GS05].

La vérification : des mécanismes de contraintes et de contrats peuvent être appliqués sur une architecture à composants afin de permettre la validation dans divers domaines [Beu+99 ; SJ13], ce qui facilite la mise au point d'un système « correct par construction » [GS05].

4.3 Caractéristiques des composants

4.3.1 Interfaces de communication

Les différents composants qui constituent un système doivent pouvoir communiquer entre eux afin d'échanger les informations nécessaires à leur réalisation. Pour permettre cette communication, il est primordial de définir la nature des échanges, le rôle de chaque composant au cours de l'échange, ainsi que le moment où la communication intervient. Deux stratégies sont couramment utilisées :

Les interfaces : chaque composant publie une liste d'interfaces qu'il implémente (interfaces fournies), ainsi qu'une liste d'interfaces requises [BD07]. La communication se fait par des appels fonctionnels de ces interfaces. Ce type de communication suit le paradigme *client/serveur* [NBP13] : un composant client adresse une requête au serveur à travers une interface, et attend un résultat en réponse.

Les flux de données : chaque composant propose des entrées et sorties, qui correspondent pour les entrées aux données nécessaires à l'exécution du composant, et pour les sorties aux données produites par le composant. La communication entre les composants se fait alors en connectant les entrées aux sorties à l'aide de connecteurs. Ce type de communication peut suivre les paradigmes *publish/subscribe* (le producteur de la donnée publie une valeur sur sa sortie, tous les abonnés reçoivent cette valeur [And+05]) ou *request/reply* (le consommateur demande une donnée, puis la reçoit de la part du producteur [San+17]).

Ces deux approches ne sont pas exclusives, certains modèles à composants proposant à la fois une communication par interface et une gestion des entrées/sorties de données. Dans les deux cas, les liens sont construits à l'aide de connecteurs, qui relient ensemble les ports de communication des composants. Selon les modèles, le niveau de complexité des connecteurs varie [Hoš+10] : ils peuvent être de simples liens, ou bien avoir leur propre comportement, souvent afin de réaliser des tâches d'adaptation et de synchronisation [BBS06], ou de contrôler l'exécution des composants [DTL15 ; LT12]. Dans certains cas, les connecteurs sont considérés comme des types spéciaux de composants [ECB06].

Le paradigme de flux de données est souvent privilégié pour la modélisation de systèmes physiques, car il est proche de la vision du système des automaticiens. La communication orientée interface est plus classiquement utilisée pour le logiciel. Cependant, le concept d'interface ne se limite pas à des appels fonctionnels entre les composants, il peut être étendu en englobant des notions plus larges [BD07] :

- l'échange d'événements (déclenchement, synchronisation) ;
- la configuration du composant, via l'édition de paramètres internes ;
- la publication de l'état du composant, si celui-ci est observable ;
- l'initialisation ou la gestion du cycle d'exécution du composant ;
- la réflectivité, qui permet d'aller interroger le composant sur son état [NBP13] ;
- la formalisation de services offerts ou requis, dans le cadre d'une architecture orientée services [DCL08 ; ECB06].

4.3.2 Comportement

Un composant encapsule une fonction du système au travers d'un comportement [BD07]. On parle également de services offerts par le composant [Rom+10 ; DCL08]. Le comportement peut résulter de l'activité du composant (s'il en dispose), d'une réaction à un événement (nouvelle valeur sur une entrée, synchronisation externe...), ou bien d'une combinaison de ces deux éléments.

Le comportement d'un composant logiciel est généralement décrit à l'aide d'un langage de programmation spécifique au domaine d'application du composant. Dans les systèmes embarqués, le C et le C++ restent les principaux langages utilisés. Le langage choisi doit proposer une sémantique opérationnelle qui intègre les aspects de communication (gestion des entrées / sorties et des interfaces du composant) décrits précédemment. Dans ce cas, il est fréquent que le comportement interne soit découplé de la couche d'interaction [GS05 ; BBS06]. Le comportement de la couche d'interaction peut être catégorisé ainsi [ECB06] :

Les composants passifs réagissent uniquement à des sollicitations extérieures : un événement sur l'une des entrées du composant entraînera la production d'une action ou d'un résultat.

Les composants actifs disposent d'une activité propre qui s'exécute périodiquement, indépendamment de leurs entrées/sorties. Dans ce cas, les données sont généralement bufferisées lors de leur production.

Les composants mixtes disposent d'une activité propre, mais répondent également à des événements externes, en lien avec leur activité.

Pour ce qui est du comportement interne, on retient que la notion d'**activité** [And+05] désigne un processus temporel périodique interne au composant. On lui associe souvent une notion de fréquence d'exécution. La temporalité de l'activité peut être gérée par le composant lui-même, ou bien être déléguée à un cadencement externe, via une interface de synchronisation ou de déclenchement. Son exécution peut être périodique ou apériodique [ECB06].

4.3.3 Propriétés

Les composants peuvent disposer d'un ensemble d'attributs qui leur sont propres. On parle dans ce cas de propriétés. Les propriétés peuvent être éditables et/ou observables depuis l'extérieur du composant, soit au travers d'une interface, soit par un mécanisme dédié du modèle à composants.

On distingue les attributs qui permettent de configurer le comportement du composant (on parle souvent de paramètres) [San+17 ; NBP13] des attributs qui contrôlent le cycle de vie du composant et qui permettent de connaître son état [And+05]. Il est également fréquent de trouver

des attributs pour définir les conditions initiales du comportement et les périodes d'activation [BD07].

4.3.4 Structure

Lorsque plusieurs composants sont assemblés entre eux pour former un système cohérent, on obtient une **configuration**. Celle-ci peut être décrite par plusieurs moyens : langages de script, fichier de balisage (XML), langage dédié, outil graphique [DTL15 ; And+05]. Changer la configuration du système revient à ajouter, supprimer ou remplacer un ou plusieurs composants, ou encore les liens qui les relient.

La modélisation d'un système complexe est plus simple lorsqu'il est possible de structurer les composants au sein d'une configuration. Il s'agit principalement d'organiser les composants en groupes au sein d'un ensemble cohérent, que l'on désigne sous le terme de conteneur. Plusieurs types de structuration sont possibles, selon la nature du conteneur.

L'approche la plus simple consiste à considérer le conteneur comme un bloc contenant un ensemble de composants, sans apporter de fonctionnalités supplémentaires. Les composants sont regroupés en sous-systèmes de manière cohérente [San+17], mais les connexions ne sont pas affectéees par cette répartition, la structure n'impacte pas le comportement.

Il est également possible d'avoir des conteneurs qui respectent les principes du *design pattern* composite [Gam+95] :

> « Composite lets clients treat individual objects and compositions of objects uniformly »

Le conteneur offre alors les mêmes services qu'un composant : des interfaces, des ports de communication, et éventuellement une activité. On confond alors conteneurs et composants. Il en résulte deux types de composants [And+05] : les **composants atomiques**, qui constituent les briques élémentaires de la structure, et les **composants composites**, qui contiennent des composants. Les composites sont traités comme des composants normaux [Rom+10 ; DCL08].

Les composants composites peuvent disposer d'un comportement propre, ou bien simplement encapsuler le comportement de leurs constituants. On peut trouver deux types de connecteurs au sein des composites [DCL08 ; LT12] : ceux qui assurent une communication horizontale entre les composants contenus, et ceux qui permettent de publier à l'extérieur du composite une interface ou un port issu d'un composant contenu.

Dans le cas où les conteneurs peuvent contenir d'autres conteneurs, il devient possible de créer une hiérarchie de composants à plusieurs niveaux. Cela permet de modéliser le système avec une granularité variable : les premiers niveaux de la hiérarchie modélisent la structure générale du système, tandis que les niveaux inférieurs détaillent le fonctionnement interne des sous-systèmes. Cette hiérarchie est principalement utile lors de la conception de l'application, et donc présente surtout dans les systèmes de composants de développement.

4.3.5 Types et instances

Comme classiquement dans le paradigme orienté objet, il y a une distinction entre le type d'un composant et une instance de ce composant. Généralement, le type est défini par l'ensemble des interfaces et entrées/sorties du composant, par la nature de ses propriétés, et par une description de son comportement [LT12]. Pour pouvoir utiliser le composant au sein d'une configuration, il est nécessaire d'en créer au moins une instance. Selon les types de systèmes de composants et les frameworks utilisés pour les manipuler, plusieurs approches sont envisageables pour gérer le mécanisme d'instanciation lors de la création d'une configuration :

- L'utilisation de types : cela implique de créer ou de disposer d'une bibliothèque de types de composants [LLE07]. Plusieurs instances du même type peuvent être créées, ce qui facilite la réutilisation des définitions des composants [Gen+02]. Si le comportement du composant est défini dans son type, la création d'une configuration se résume à un paramétrage permettant de lier entre eux les composants.
- L'équivalence entre type et instance : le type du composant est créé lors de la définition de l'instance, et peut éventuellement évoluer de manière dynamique [DGR09]. On obtient alors

un singleton (une seule instance par type) pour chaque composant. La réutilisation se base sur une copie d'un composant déjà instancié. Ce mécanisme permet une grande souplesse lors de la création d'une configuration, chaque composant pouvant être adapté à la fonctionnalité précise qu'il doit remplir. Cependant, une adaptation du comportement de chaque singleton est nécessaire pour prendre en compte de manière effective son interface.

– Le passage entre type et instance peut se faire de manière progressive, en définissant peu à peu les attributs du composant, selon l'approche *clabject* [GH06]. A chaque étape, le type se raffine et peut être utilisé pour décrire une part spécifique des instances. Quand tous les attributs sont définis, on obtient une instance.

Afin de profiter des différents avantages de ces approches, celles-ci peuvent être combinées entre elles : la plupart des frameworks de modèles à composants proposent des outils pour la création et l'utilisation de bibliothèques de composants permettant leur réutilisation [LLE07 ; DTL15], certains offrent également la possibilité de créer des singletons, en définissant le type de manière *ad-hoc* et implicite, ce qui peut être particulièrement intéressant pour des composants n'ayant pas de comportement propre (composants de structuration).

4.4 Mise en œuvre des composants

Une fois le système de composants défini, il est nécessaire de lui donner une sémantique d'exécution pour pouvoir le mettre en œuvre. Cette partie présente les différentes questions qui se posent lorsque l'on veut mettre en œuvre les composants de développement pour la simulation. Ainsi, les problématiques spécifiques au développement de systèmes embarqués temps-réel, qui concernent les composants d'exécution, ne sont pas détaillées ici.

4.4.1 Aspects temporels

Les modèles de systèmes cyber-physiques représentent des éléments qui évoluent au cours du temps. Les composants de ces systèmes doivent intégrer une modélisation du temps pour pouvoir s'exécuter [Lee15]. Bien que le temps puisse être représenté de manière continue, il est généralement discrétisé dans les approches logicielles. Cette section s'intéresse aux différentes stratégies de gestion du temps qui sont envisageables pour les composants des systèmes cyber-physiques.

L'aspect temporel des communications

Une stratégie temporelle doit être définie pour que l'échange d'information puisse avoir lieu entre plusieurs composants. Selon la nature de la communication, cette stratégie peut prendre diverses formes [BBS06].

– Dans le cas d'une relation client/serveur, la temporalité est gérée par le client, qui envoie des requêtes au serveur. Cependant, ces appels peuvent être bloquants (le client cesse son activité le temps de recevoir la réponse), ou non bloquants [Hoš+10].

– Le paradigme *publish/subscribe* peut utiliser plusieurs stratégies : une seule publication, publication à fréquence fixe, publication dès qu'une nouvelle donnée est produite, etc. [And+05 ; San+17].

Gestion du temps à l'exécution

Plusieurs possibilités sont envisageables pour modéliser le temps lors de l'exécution. Dans la plus simple des approches, l'exécution d'un pas de temps d'un composant se fait sans interruption, et donc sans interaction avec le reste de la configuration (on parle de paradigme *run to completion*) : le composant dispose de toutes les données d'entrée dont il a besoin avant le début de l'exécution, il produit les sorties à la fin de celle-ci. Le non-respect de ce paradigme permet une gestion plus fine des priorités d'exécution et de gestion des contraintes temps réel. Cependant, de telles approches sont complexes et nécessitent la mise en œuvre d'un ordonnanceur préemptif.

La gestion du temps peut être déléguée à une horloge centralisée qui vient activer périodiquement les composants, elle peut également être distribuée (dans le cas de plusieurs plateformes

matérielles par exemple). Des mécanismes de synchronisation des horloges et d'ordonnancement sont alors à prévoir si l'on veut pouvoir garantir des propriétés temps réel [Des+17]. Lorsque cette ou ces horloges avancent au rythme du temps physique, on parle d'exécution temps réel. Dans le cas inverse, on parle de temps simulé. L'exécution temps réel implique la mise en place de mécanismes d'analyse pour s'assurer que l'exécution du comportement d'un composant peut s'effectuer entre deux pas d'activation du composant [Mas+14].

4.4.2 Ordonnancement

Lorsque des composants s'exécutent de manière concurrente (partage de ressources, échange d'informations...), la question de l'ordonnancement se pose. Il s'agit de savoir dans quel ordre les composants vont être exécutés. Plusieurs stratégies peuvent être mises en œuvre pour résoudre ce problème :

Ordre défini explicitement : le créateur de la configuration indique explicitement soit un ordre d'exécution des composants, soit un niveau de priorité pour chaque composant [BBS06]. Cette approche est simple et efficace car elle permet de garder la maîtrise de l'exécution. Elle requiert cependant une charge de travail supplémentaire lors de la modélisation.

Ordre défini par le flux de données [Bil+96] : l'ordre d'exécution est dicté par les dépendances des composants entre eux, afin de s'assurer qu'un composant dispose de toutes les données requises avant de s'exécuter. Cet ordre peut être déterminé lors de l'initialisation, ou bien en amont, lorsque la configuration est connue. Cette méthode permet d'alléger la charge de travail nécessaire à la création d'une configuration. Cependant, les dépendances temporelles ne sont pas toujours connues à l'avance, et la méthode peut être mise en échec par la présence de dépendances cycliques.

Ces stratégies peuvent se combiner afin de pallier leurs lacunes respectives : l'ordre du flux de données est respecté et un mécanisme de priorités permet de résoudre les cycles de dépendances.

4.4.3 Initialisation

Un mécanisme d'initialisation peut-être proposé par le système de composants afin de permettre l'exécution des composants dans de bonnes conditions [ECB06 ; LT12] . Il est fréquent que les services et fonctionnalités proposés par le composant ne soient disponibles qu'après l'initialisation. Dans ce cas, l'architecture du système de composants doit garantir la bonne initialisation de l'ensemble de la configuration avant de débuter l'exécution du système étudié.

La notion d'initialisation est très liée au type de composant ainsi qu'à la nature des opérations qu'il réalise. Elle peut être nécessaire pour initialiser les propriétés des composants (paramètres, conditions initiales), mais également pour initialiser les liaisons entre composants (échange de données/métadonnées concernant les politiques de communication, découverte de la configuration à l'exécution...) [NBP13].

Selon le type de modèle de composant, l'initialisation peut intervenir avant l'exécution, pour préparer le composant, et peut également servir à réinitialiser le comportement de celui-ci pendant l'exécution.

4.4.4 État du composant

La notion d'initialisation peut être étendue par des notions de démarrage et de finalisation, afin d'assurer la disponibilité et la restitution des ressources nécessaires aux composants [ECB06].

Certains systèmes de composants proposent une gestion complète du cycle d'exécution du composant : une machine à états tient à jour l'état du composant, et donc sa capacité à interagir et à exécuter son comportement [And+05]. Selon la complexité du système de composants, plusieurs états peuvent être proposés : initialisé, en attente, en erreur, finalisé... L'état du composant peut être observable depuis l'extérieur, un changement d'état pouvant servir au déclenchement d'actions sur d'autres composants.

4.5 Discussion sur les modèles à composants

Un système de composants peut se présenter sous de multiples formes. Ce chapitre a permis de synthétiser les questions qui se posent lors de la conception d'un tel système. Les choix effectués en termes de stratégie de communication et de structuration sont souvent dictés par l'utilisation attendue du système de composants et le contexte dans lequel il va être utilisé. Ces choix portent sur :

- un paradigme de communication,
- un modèle de description du comportement,
- un mécanisme d'attributs et de propriétés des composants,
- un modèle de structuration,
- un mécanisme de typage des composants,
- un modèle d'exécution temporelle (ordonnancement) et de gestion des états des composants.

Pour modéliser un système cyber-physique, les composants de développement semblent plus indiqués, car le système visé n'est pas entièrement logiciel. Dans la plupart des cas, une telle modélisation est utilisée à des fins de documentation, de vérification [Beu+99 ; Mah13 ; SJ13] et de génération de code [Vid+09 ; Las+09]. La souplesse des systèmes de composants permet également de simuler le système modélisé, à différents niveaux : soit l'ensemble du système est modélisé par des composants logiciels lors de la simulation, soit certaines parties physiques du système (capteurs, contrôleur, système mécanique...) sont intégrées en tant que composants lors de la simulation. Il est envisageable de remplacer progressivement les composants de simulation par des composants réels pour obtenir finalement le système complet.

Pour adapter un système de composants aux besoins de la modélisation et de la simulation d'un système cyber-physique, il est nécessaire de faire des choix parmi les différentes possibilités exposées ci-dessus, en termes de structure, de stratégie de communication et de description du comportement. Parmi les systèmes de composants existants, certains sont très généralistes et permettent l'implémentation de plusieurs stratégies, d'autres sont plus spécialisés. Il est donc possible soit de spécialiser un système généraliste pour l'adapter aux contraintes des systèmes cyber-physiques, soit de réutiliser un système spécialisé proche de la problématique à traiter. Le risque dans le premier cas est d'avoir un système lourd et complexe dont la plupart des fonctionnalités ne seront pas utilisées, tandis que dans le second scénario, le risque est d'être bridé dans la réutilisation du système existant, à cause de sa trop forte spécialisation. La troisième option est de créer un système de composants dédié, qui intègre les fonctionnalités nécessaires à la modélisation du système et qui contient les outils adaptés au domaine d'étude. Cela permet de prendre en compte les spécificités en termes d'hétérogénéité, et d'utiliser les paradigmes de communication, de structuration et d'intégration usuels dans le domaine.

Chapitre 5

Architecture logicielle pour le contrôle et la simulation de systèmes cyber-physiques

La réalisation d'un simulateur de système cyber-physique doit être facilitée par une approche IDM. Dans ce cadre, l'utilisation d'un modèle à composants permet de structurer le logiciel, mais ne donne aucune information sur l'architecture du simulateur et la manière dont doivent être agencés les composants entre eux. Il est donc nécessaire de définir une architecture pour notre système, qui soit à la fois compatible avec son domaine d'application, à savoir le contrôle, et avec les spécificités liées à la simulation.

Par la suite, vu que nous cherchons à simuler un système de contrôle, nous étudions d'une part les architectures qui sont dédiées aux contrôleurs de systèmes cyber-physiques, et d'autre part les architectures destinées à la simulation de ces systèmes.

5.1 Architectures pour le contrôle

Les architectures logicielles pour les contrôleurs de systèmes cyber-physiques sont variées et souvent complexes. Depuis la fin des années 1960, de nombreux modèles d'architecture ont été développés [Pas+14]. L'objectif de cette partie est de proposer un aperçu de ce qui se fait en termes d'architecture dans le domaine considéré. Pour cela, différents styles architecturaux sont étudiés en fonction des critères suivants : l'intention de l'architecture, la présence d'un modèle à composants, l'utilisation de l'IDM, les différentes couches du style architectural, la plateforme et les technologies (langages) supportées par l'architecture, ainsi que son caractère générique ou spécifique.

Sans prétendre à l'exhaustivité, il s'agit ici de présenter un ensemble qui apparaît comme représentatif des différentes solutions pour la mise au point et la simulation de systèmes cyber-physiques. On ne s'intéresse pas ici aux approches dédiées à la conception robotique, à la gestion de la tolérance aux fautes et au déploiement.

5.1.1 MontiArc

MontiArc [HRR14] est un framework permettant de développer des logiciels de contrôle pour robots par une approche dirigée par les modèles. Il contient un langage de description d'architecture (ADL), des outils de modélisation et de simulation pour les architectures, ainsi que divers générateurs de code.

Les architectures sont décrites en utilisant un modèle à composants, ceux-ci pouvant être regroupés sous forme de packages ou de composants composites. Par ce mécanisme, le framework n'impose pas un style architectural particulier, il met à disposition des outils de structuration de l'architecture. Les communications entre les composants sont assurées au travers de ports de données, ces dernières étant typées. Les communications s'effectuent de manière unidirectionnelle et

asynchrone.

Le langage de description d'architecture MontiArc est un DSL textuel. Il est basé sur le framework MontiCore [KRV10] qui permet la création de DSL selon une approche dirigée par les modèles. MontiCore permet ainsi la définition d'une sémantique (équivalente à un méta-modèle) ainsi que d'une grammaire concrète pour le langage MontiArc.

MontiArc est un langage de description d'architecture généraliste, mais qui propose des mécanismes d'extension permettant de le spécialiser pour répondre à des contraintes spécifiques aux domaines visés [But+17]. Ces extensions peuvent concerner la modélisation et l'intégration du comportement des composants, les vérifications et transformations appliquées aux modèles et la génération de code. Par exemple, l'extension MontiArcAutomaton [RRW14] permet la description du comportement des composants par des automates à états. Elle propose également des générateurs de code pour plusieurs technologies : le langage MONA permettant de l'analyse formelle, les langages Java et Python pour le déploiement de l'architecture, mais également le méta-modèle *Ecore* d'EMF pour permettre une édition graphique de l'architecture.

5.1.2 ORCCAD

ORCCAD (*Open Robot Computer Aided Design*) est un environnement de programmation pour les systèmes robotiques qui permet la conception d'architectures pour les logiciels dédiés aux contrôleurs, en intégrant nativement la notion temps réel [Bor+98]. Les architectures sont implicitement découpées en couches par l'approche suivante : des *Robot-Tasks* représentent les actions robotiques élémentaires et encapsulent les lois de contrôle bas-niveau, tandis que les *Robot-Procedures* sont en charge des missions plus complexes : elles sont composées de plusieurs *Robot-Tasks*, et éventuellement de *Robot-Procedures*.

A l'origine, l'environnement ORCCAD se présente sous forme d'une boîte à outils composée de classes C++. Il a par la suite été adapté aux méthodes de l'IDM via le Framework EMF d'Eclipse [Ari+10]. Des outils d'édition textuels et graphiques, basés sur GMF, sont proposés. La génération de code à partir des modèles se fait via le générateur Xpand.

ORCCAD a une portée générique dans le domaine de la robotique, et plus largement des systèmes contrôlés par ordinateurs. L'outillage d'ORCCAD permet l'analyse et la vérification de propriétés logiques et temporelles pour les *Robot-Tasks* et les *Robot-Procedures*. La génération de code exécutable (C++/ESTEREL) est également possible, l'environnement ORCCAD permet ainsi de produire des logiciels de contrôle en s'appuyant sur les API Linux ou sur Xenomai. Il a pu être expérimenté et validé sur différents types de robots : des bras articulés, des véhicules autonomes, des robots marins et des drones. L'environnement ORCCAD se révèle donc efficace pour la conception d'architectures pour les logiciels embarqués, mais ne présente pas, en revanche, de mécanisme dédié à la simulation.

5.1.3 PROTEUS

PROTEUS (*Plateforme pour la Robotique Organisant les Transferts Entre Utilisateurs et Scientifiques*) est un projet de recherche ANR ayant pour but de faciliter les transferts technologiques entre les problématiques de la robotique et l'ingénierie logicielle [MP08]. Le projet définit au travers de modèles les concepts de base pour la robotique, et ce à plusieurs niveaux d'abstraction [LDG10] :

- à un haut niveau, nommé *kernel*, on trouve les notions d'architecture et de composants. Il est à noter que les composants peuvent être logiciels ou matériels. Il est intéressant de remarquer que l'environnement est défini comme une entité spécifique.
- à un niveau générique, on retrouve les composants classiques du domaine et des applications de contrôle : les éléments contrôlables, les capteurs et les actionneurs.
- à un niveau métier, on trouve des composants spécifiques en lien avec le domaine d'application du robot.

L'approche de PROTEUS est à vocation générique : elle n'impose pas de modèle de composants particulier ou de plateforme d'exécution spécifique. Si elle a permis l'émergence de plusieurs DSL permettant de décrire les solutions métier à un problème robotique, elle reste dans la définition générale d'un cadre de conception.

5.1.4 CLARAty

CLARAty (*Coupled Layer Architecture for Robotic Autonomy*) [Vol+01] est un modèle d'architecture conçu pour assurer la modularité des systèmes de contrôle autonomes. Il s'agit d'une architecture mixte basée sur deux couches : une couche fonctionnelle, orientée objet, et une couche décisionnelle, pouvant être déclarative ou procédurale, qui utilise l'abstraction de la couche fonctionnelle pour initier et accomplir les différentes tâches du robot.

Trois niveaux d'hétérogénéité [Nes+06] sont considérés dans les systèmes de contrôle : la configurations des capteurs, les capacités du système contrôlé, et l'architecture matérielle du système de contrôle. La gestion de cette hétérogénéité passe par des composants d'adaptation, qui produisent des sorties identiques quelles que soient la ou les technologies sous-jacentes des capteurs. CLARATy n'utilise pas explicitement de modèle à composants, mais une approche orientée objet, dont les mécanismes d'héritage permettent de représenter le système à différents niveaux d'abstraction et de granularité. Cette approche orientée objet exploite les possibilités d'UML à des fins de conception et de documentation.

L'implémentation de CLARATy est réalisée dans le langage C++ qui offre un support total du paradigme orienté objet tout en étant compatible avec les besoins temps réel. Cette implémentation est déployable sur les plateformes VxWorks, Linux et Solaris.

CLARATy propose également des outils de configuration et des interfaces d'accès aux différents niveaux de contrôle du système à des fins de simulation, ce qui permet d'explorer divers scénarios. Pour assurer la généricité et la modularité, des interfaces standards sont définies à différents niveaux d'abstraction. La flexibilité se fait alors au détriment de la simplicité : la généricité requiert une complexification des interfaces standard.

5.1.5 MAUVE

MAUVE (*Modeling AUtonomous VEhicles*) est un DSL basé sur un modèle à composants qui suit une approche dirigée par les modèles. Il permet, via l'instanciation de composants, de décrire l'architecture d'un logiciel temps réel pour la robotique. L'objectif premier de MAUVE est la validation de propriétés temps réel par une analyse temporelle de l'architecture [LDC11].

Les composants sont constitués d'une coque (*shell*) et d'un cœur (*core*) [GLD14]. La coque définit les interfaces du composant, pouvant prendre la forme de propriétés, de ports de données (qui possèdent un type et une orientation) et d'opérations permettant d'appeler des fonctions ou d'envoyer des requêtes au composant. Le cœur définit le comportement du composant, soit en établissant une correspondance entre les opérations du composant et des fragments de code C++ (*codels*), soit en définissant une machine à états à partir d'un ensemble d'états et de transitions.

En complément du DSL, MAUVE fournit des outils afin d'assurer le déploiement et l'exécution du logiciel : ces outils permettent de générer du code pour le *middleware* Orocos, lequel peut être enrichi par le code métier des *codels*. Le code ainsi obtenu est instrumenté afin de réaliser des mesures du temps d'exécution. Un script de déploiement des composants Orocos est également généré, le résultat peut directement être exécuté sur la platforme cible.

MAUVE dispose également de son propre environnement d'exécution, *MAUVE runtime* [DGL17] implémenté en C++. Celui-ci permet d'outrepasser certaines limitations du *middleware* Orocos en termes de synchronisation, de partage des ressources ou encore de reconfiguration des composants à l'exécution.

Les couches DSL et *runtime* de MAUVE ont été déployées et testées sur plusieurs robots mobiles embarquant des capteurs divers (odomètre, scanner laser, caméra) dans l'objectif de réaliser diverses fonctions de navigation : localisation, cartographie, suivi de trajectoire, planification... Des composants spécifiques à ces applications ont ainsi été développés et intégrés dans MAUVE.

5.1.6 SAIA

SAIA (*Sensors Actuators Independent Architecture*) [DeA07] est un style architectural adapté aux systèmes dédiés au contrôle de procédés, systèmes qui sont étroitement en lien avec leur environnement. Son objectif est d'assurer l'indépendance du contrôle vis-à-vis de la plateforme sur laquelle il est exécuté. Cela permet d'intégrer la variabilité des plateformes due à leurs évolutions,

mais également de disposer de plateformes de simulation pour évaluer l'application de contrôle.

SAIA propose l'introduction d'un modèle de plateforme abstraite, nommé SAIM (*Sensors / Actuators Independent Model*), qui dispose des entrées/sorties nécessaires au contrôle, indépendamment de toute technologie de capteurs et d'actionneurs. D'un autre coté, la plateforme réelle est modélisée dans une couche nommée SAM pour *Sensors Actuators Model*. Ces deux couches sont reliées par un connecteur réalisant l'adaptation des données qui y transitent. Ce dernier est spécifié dans l'ALM (*Adaptation Layer Model*).

L'ALM utilise de manière intensive les composants logiciels en faisant usage d'un modèle à composants *ad-hoc* réalisé spécialement pour SAIA : le connecteur reliant la couche SAIM et la couche SAM est un composant composite constitué d'un assemblage de sous-connecteurs, chacun lié à un type d'entrée ou de sortie. Ce sont ces sous-connecteurs qui sont en charge des fonctions d'adaptation, d'interprétation, de formatage et de conversion des informations. La qualité de service est également gérée par l'ALM, les contrats qui y sont liés pouvant être adaptés par le connecteur.

Les différents modèles de SAIA peuvent être manipulés via des outils de modélisation basés sur le framework GMF [DB06]. Un démonstrateur a été réalisé sous la forme d'un robot d'exploration, dont seuls les aspects temporels du comportement ont été modélisés. La modélisation des différentes couches a permis de générer du code dans le langage IF [BGM02], spécialisé dans les automates temporisés, pour effectuer des vérifications en termes de qualité de service. En assurant l'indépendance entre plateforme matérielle et logiciel de contrôle, SAIA offre donc une gestion de l'hétérogénéité des plateformes et met en place les mécanismes permettant la simulation des systèmes de contrôle. Cependant, seuls les aspects temporels sont considérés dans cette approche.

5.1.7 IMOCA

IMOCA (*archItecture for MOde Control Adaptation*) [GB13] est une architecture spécialisée pour les systèmes de contrôle de processus qui évoluent dans des environnements naturels perturbés susceptibles de remettre en cause l'intégrité physique du processus. Intégrant SAIA, l'architecture IMOCA est constituée de trois couches : la cible, l'interprétation et le contrôle. La cible modélise la plateforme matérielle (capteurs et actionneurs), tandis que l'interprétation fournit au contrôle une vue idéale de l'environnent et du système contrôlé. Le rôle de la couche d'interprétation est donc d'opérer l'adaptation des données qui circulent entre la cible et le contrôle. Ainsi, une donnée transmise au contrôleur peut être issue de plusieurs signaux de capteurs, qui auront été filtrés, formatés et éventuellement combinés par la couche d'adaptation.

Dans l'architecture IMOCA, la couche de contrôle est elle-même composée de trois agents : un agent *réactif*, qui assure le maintien d'une loi de contrôle, un agent *adaptatif*, qui vient modifier les paramètres de la loi de contrôle en fonction du contexte, et un agent *expert*, sous forme d'une machine à états, qui change de loi de contrôle quand des transitions, liées au contexte, sont déclenchées. On parle alors de changement de mode de contrôle.

La séparation en couches et les différents agents du contrôleur sont formalisés dans le méta-modèle IMOCA, exprimé à l'aide du framework EMF. Un second méta-modèle, ImocaGen [GB16] propose des outils permettant la génération d'un code qui respecte l'architecture IMOCA. Plusieurs langages peuvent être générés (java, C, *C-like*), permettant ainsi d'adresser de multiples plateformes. La structure de l'application de contrôle est ainsi générée, différents mécanismes permettent l'intégration de code lié à la plateforme visée et de code métier spécifique au domaine d'application.

L'architecture IMOCA est spécialisée pour les systèmes qui interagissent fortement avec leur environnent. Elle propose ainsi une méthodologie permettant de choisir les modes de contrôle pertinents pour de tels systèmes, et d'optimiser le paramétrage de ces modes. Elle a été testée sur des systèmes physiques comme un char à voile miniature dont on cherche à conserver le cap et l'assiette, et également sur des systèmes simulés dans le but de mettre au point des lois de pilotage pour les voiliers [GB13]. Par les mécanismes de génération de code, IMOCA fournit une Interface Homme Machine (IHM) permettant d'ajuster les coefficients des lois de contrôle, mais l'architecture ne propose pas d'outils autre que l'IHM qui soient dédiés à la gestion de la simulation.

5.1.8 Aerostack

Aerostack est à la fois un modèle d'architecture et un framework destiné aux drones aériens (UAS - *Unmanned Aerial Systèmes*) [San+16]. L'objectif de l'approche est double : permettre un haut degré d'autonomie lors des missions des UAS, tout en restant indépendant d'un matériel et d'une plateforme spécifiques. L'architecture d'Aerostack est dite hybride car certains de ses constituants sont délibératifs et d'autres sont réactifs. Pour ce faire, elle propose cinq couches distinctes :

Une couche réactive qui contrôle les entrées/sorties à partir de boucles d'asservissement.

Une couche délibérative qui gère la mission dans son ensemble en générant une séquence d'actions pour répondre à des tâches complexes.

Une couche exécutive qui convertit les actions de la couche délibérative en des séquences de comportements pour la couche réactive.

Une couche réflective qui supervise les autres couches et permet la détection de problèmes éventuels.

Une couche sociale en charge de la communication avec d'autres UAS ou avec des opérateurs.

On retrouve dans les trois premières couches un découpage des niveaux d'abstraction qui permet de s'abstraire de la plateforme dans les plus hauts niveaux du contrôle, de manière similaire aux approches de SAIA et d'IMOCA. Pour mettre en place cette architecture, Aerostack se base sur un modèle à composants : chaque couche est constituée d'un assemblage de différents composants, ceux-ci étant définis à divers niveaux d'abstraction. Par ailleurs, le framework dispose de plusieurs bibliothèques de composants spécialisés dans le domaine des UAS. On y trouve des composants bas niveau pour gérer un certain nombre de capteurs et d'actionneurs, des composants plus abstraits permettant de faire de la commande et de la planification, des composants contenant des algorithmes spécifiques au domaine, des composants permettant l'interfaçage avec le middleware ROS [ROS19], ou encore des composants permettant la réalisation de modules de simulation.

Grâce à la richesse de ses bibliothèques, la création d'une application de contrôle en utilisant le framework Aerostack peut se faire à moindre effort [San+17] : la création de l'architecture se fait via l'écriture de *launch files* contenant les appels aux composants, leur configuration passe par l'édition de fichiers XML. La planification de mission se fait également par le biais de l'édition de fichiers XML.

Le framework a été testé à plusieurs reprises, y compris dans des situations de collaboration entre plusieurs drones. Un certain nombre de métriques sont proposées pour évaluer les gains d'une telle architecture. Si la simulation est possible à l'aide du framework Aerostack, la modélisation et l'intégration de l'environnement (et en particulier du vent, qui a une forte influence sur les drones aériens) n'ont pas fait l'objet d'effort particulier à notre connaissance.

5.2 Architectures pour la simulation

La simulation d'un système comportant plusieurs éléments pouvant s'exécuter en parallèle, répartis en diverses couches et éventuellement distribués sur plusieurs plateformes implique de relever divers défis. Il faut pouvoir exécuter chaque élément, les faire communiquer entre eux, gérer leur synchronisation et leur ordonnancement. Selon l'architecture du simulateur, les différents composants peuvent être fortement hétérogènes en termes de nature (composant logiciel, système physique, modélisation complexe, agent de simulation indépendant...) et de plateforme d'exécution (système distribué).

La réalisation d'un simulateur « monolithique » spécialement dédié au besoin du système simulé est toujours possible. Cependant, cette solution demande souvent un effort significatif de conception et de développement, et la maintenance est rendue difficile par la complexité du système. L'évolution du simulateur demande un fort travail d'intégration de nouveaux composants. Afin d'éviter ces inconvénients, nous étudions ici des architectures génériques permettant la conception modulaire de simulateurs pour faciliter l'intégration de leurs divers constituants.

5.2.1 HLA

High Level Architecture (HLA) est une spécification proposée à la fin des années 1990, visant à l'origine à unifier les architectures dédiées à la simulation pour le *US Department of Defense* [Dah97 ; DM98]. Elle a par la suite été standardisée par l'IEEE [IEE10]. Le standard définit un ensemble de règles, des spécifications d'interface et des templates d'objets permettant de mettre en place des outils de simulation.

L'architecture préconisée par HLA est composée de deux types d'éléments :

Les *federates* sont les unités élémentaires de simulation. Un ensemble de *federates* distribués sont regroupés en une *federation*, au sein de laquelle ils interagissent entre eux.

La *RunTime Infrastructure* (RTI) est un système d'exploitation distribué pour les *federates*, il offre divers services génériques : les interactions entre les *federates* et la gestion de ces derniers au travers de fonctions de support. Toutes les interactions entre les *federates* passent par le RTI.

L'architecture prévoit également la possibilité d'intégrer des observateurs sous forme de collecteurs de données et de systèmes de monitoring. Enfin, les interfaces d'interaction avec un utilisateur permettent d'envisager d'intervenir en direct sur la simulation.

Le standard spécifie les interfaces entre les *federates* et la *RTI*. Différents services peuvent transiter par ces interfaces, permettant entre autres :

- La gestion des *federates* (création et reconfiguration de *federation*)
- La gestion des données (déclaration, distribution durant la simulation)
- La gestion des objets (création, suppression, édition d'attributs et changement de propriétaire)
- La gestion du temps (synchronisation)

HLA est un standard dédié à la simulation intensive distribuée, il n'impose pas d'implémentation particulière et reste indépendant d'une technologie ou d'un langage de programmation. Les *federates* peuvent être de natures diverses (ordinateur, système d'interaction avec l'utilisateur, enregistreur de données, simulateur séparé). La gestion de cette hétérogénéité est permise par le respect de la spécification de l'interface *federate*/RTI. En dehors de cette interface, HLA n'impose aucune contrainte sur ce qui doit être contenu dans les *federates*, et n'offre pas de librairies spécifiques pour le domaine visé.

L'architecture HLA est adaptée à la simulation de gros systèmes nécessitant des capacités de calcul importantes, et pour lesquels la performance représente un enjeu majeur. La complexité de mise en œuvre d'une telle architecture réduit son intérêt pour les systèmes de simulation non distribués, lorsque la performance n'est pas critique.

5.2.2 FMI

Functional Mock-up Interface (FMI) est un standard indépendant d'un outil ou d'un langage permettant à la fois l'échange de modèles et la co-simulation de modèles dynamiques [FMI18]. FMI s'appuie sur des fichiers XML et du code C compilé. La première version FMI 1.0 a été publiée en 2010 [Blo+11], suivie en 2014 par FMI 2.0.

L'objectif de FMI est de fournir une interface standard pour associer deux ou plusieurs outils de simulation dans un même environnement de co-simulation. L'échange de données entre sous-systèmes est ici limité à de la communication discrète point à point. Entre deux points de communication, un sous-système est considéré comme évoluant indépendamment des autres. Un algorithme *master* contrôle l'échange de données entre les sous-systèmes, considérés comme *slaves*, et synchronise l'exécution de chaque outil de simulation. La description des informations de chaque *slave*, utiles pour le *master*, est exprimée dans un fichier XML standardisé. En particulier, ce fichier inclut l'ensemble des caractéristiques du *slave* vis-à-vis des algorithmes d'avancement du *master*, le type d'entrées sorties et le rythme d'avancement du *slave*.

Un outil de simulation compatible avec FMI est appelé FMU (*Functional Mock-up Unit*). Un FMU peut être décrit dans divers langages de programmation (C, C++, Matlab, Java, ...) offrant ainsi une capacité d'interopérabilité de langages à l'approche FMI. Le fichier XML joue ici le rôle de langage pivot pour assurer cette interopérabilité.

En particulier, les types de données sont standardisés et chaque *slave* doit implémenter une fonction *init()*, qui permet d'initialiser le FMU, et une fonction *doStep()*, qui décrit le comportement du FMU pour une étape de simulation. Via ces fonctions, le *master* est en charge d'initialiser les FMU, puis de conduire la simulation par appels successifs aux fonctions *doStep()*.

FMI a l'avantage d'être multi-langages. Il offre un cadre de développement unifié pour l'intégration de composants. Il s'appuie sur une communauté importante afin de proposer des facilités d'intégration pour les diverses plateformes de simulation. En revanche, FMI n'offre pas de librairies spécifiques pour le domaine visé. Il faut donc développer ses propres librairies et son propre style architectural pour l'utiliser dans le domaine d'application visé. De plus, l'approche est entièrement orientée donnée et ne permet pas des communications plus évoluées au travers d'interfaces dédiées. Enfin, FMI ne propose pas de mécanisme de gestion de scénarios de simulation : la modification de paramètres ou l'introduction d'événements dans la simulation passe par une réécriture partielle du *master*.

5.2.3 ROS et Gazebo

Gazebo [Gaz19] est un outil de simulation de robots, qui permet de simuler des populations de robots dans des environnements intérieurs et extérieurs complexes. Gazebo se concentre sur la simulation de la dynamique des robots à partir de modèles de solides rigides et de leurs liaisons [KH04]. Il peut également simuler l'environnement, les capteurs et les actionneurs. Un moteur de rendu 3D permet de visualiser l'évolution des robots dans leur environnement. La modélisation des différents éléments (robots, environnemnt, capteurs et actionneurs) se fait au travers d'un fichier de configuration XML (format SDF).

En ce qui concerne les capteurs, Gazebo propose des modèles pour les capteurs fréquemment embarqués sur les robots (GPS, centrale inertielle, laser, odomètre...). Il est par exemple possible de simuler l'absence de bruit, un bruit gaussien ou un bruit gaussien borné pour la mesure des différentes grandeurs physiques. Ces modèles peuvent être étendus à condition de développer des plugins, codés en C++, qui simulent les extensions.

Pour la simulation de systèmes contrôlés, il est possible de lier Gazebo et ROS [Tak+16]. L'idée est ici de faire du simulateur Gazebo un nœud ROS et d'utiliser ROS comme un *middleware* de simulation (dans la même idée que la RTI de HLA, mais ROS est multi-processus et non distribué).

Gazebo est un outil générique adapté à la simulation de la cinématique d'un mobile terrestre. Comme il s'agit d'un logiciel libre et soutenu par une communauté dynamique, des extensions ont été développées pour le milieu marin et sous-marin [Man+16], mais il n'existe pas à notre connaissance de modèle de voilier implémenté aujourd'hui dans Gazebo. Il est donc nécessaire de développer des nœuds spécifiques (modèle de bateau et de vent) pour pouvoir simuler un voilier dans son environnement. Le mécanisme de plugins et l'interface avec ROS permettent d'étendre facilement les possibilités de simulation, mais ajoutent une complexité temporelle lors de la simulation.

5.3 Discussion sur les architectures

Comme on le constate sur le tableau 5.1, les architectures spécialisées dans les systèmes de contrôle présentent un certain nombre de similitudes. On note tout d'abord qu'un découpage en couches est souvent présent. Ce mécanisme permet de raisonner sur le système avec divers niveaux d'abstraction. L'utilisation d'adaptateurs pour la communication entre les couches permet de faire la traduction entre ces niveaux d'abstraction. Ensuite, un modèle à composants est systématiquement présent, au moins de manière implicite, car il permet de formaliser les échanges entre les divers éléments du système. On remarque également que les outils de l'IDM sont souvent exploités dans ces architectures, soit à des fins d'analyse, soit à des fins de génération. Dans ces modèles, les architectures dédiées aux systèmes de contrôle présentent souvent un paradigme orienté donnée, même si la notion de service et d'interface complexe est parfois présente également. L'hétérogénéité des composants et des plateformes peut être prise en compte dans l'architecture à travers des notions d'adaptateurs ou de couches d'adaptation. Les approches spécialisées fournissent souvent une bibliothèque de composants liés au domaine visé. Néanmoins, ces architectures sont principalement centrées sur les contrôleurs, et la prise en compte de l'environnement dans lequel évolue le système y est souvent marginale.

	Intention de l'architecture	**Couches du style architectural**	**Modèle à composants**	**Utilisation de l'IDM**	**Plateforme Technologie**	**Spécialisation**
MontiArc	Description d'architecture de logiciels pour la robotique	Non imposé	Composants logiciels	Modélisation basée sur le DSL MontiCore	Non imposée	Générique avec possibilité d'extensions pour spécialisation
ORCCAD	Vérification et déploiement des systèmes contrôlés par ordinateur	Couche application (*Robot-Procedure*) et couche commande (*Robot-Tasks*)	Découpage en *Robot-Tasks* et *Robot-Procedure*	Adaptation à l'IDM via EMF	Linux / Xenomai	Générique
PROTEUS	Transferts technologiques robotique / ingénierie logicielle	3 niveaux d'abstraction : Kernel, générique et métier	Composants logiciels et matériels	Pas d'utilisation directe	Non imposée	Générique
CLARATy	Modularité des systèmes robotiques en autonomie	2 couches : fonctionnelle et décisionnelle	Approche orientée objets	UML pour la conception et la documentation	VxWorks, Linux ou Solaris	Générique
MAUVE	Analyse temporelle des systèmes robotiques	Non imposé	Composants logiciels	Modélisation des composants et utilisation d'un DSL	Middleware Orocos ou Mauve-runtime	Robots mobiles et problématiques temps réel
SAIA	Indépendance du contrôle vis-à-vis de la plateforme	3 couches : plateforme abstraite, réelle et adaptateur	Composants logiciels	Méta-modélisation des trois couches architecturales	Langage IF	Problématiques temps réel
IMOCA	Contrôle des systèmes dans un environnement incertain	SAIA + 3 couches de contrôle : réactif, adaptatif et expert	Composants logiciels	Méta-modélisation des couches architecturales et génération de code	Langages Java, C, C-like	Systèmes en environnement incertain
AeroStack	Contrôle de drones aériens	5 couches (réactive, délibérative, exécutive, réflective et sociale)	Composants logiciels	Pas d'utilisation directe	Configuration par fichier XML. Middleware ROS	Missions de drones aériens

TABLE 5.1 – Caractéristiques des architectures spécialisées dans les systèmes de contrôle

Les architectures spécialisées dans la simulation de systèmes présentent également des similitudes. On y trouve un découpage en blocs de simulation indépendants qui partagent des données et des informations. Elles proposent avant tout des interfaces standardisées permettant d'intégrer des éléments hétérogènes au sein d'une même simulation. Il est ainsi envisageable de combiner ces architectures entre elles, par exemple en intégrant des FMU au sein d'une architecture HLA [Bou+18], ou encore de les combiner avec les outils de l'ingénierie dirigée par les modèles [Gue+16].

Si la standardisation des interfaces de communication permet l'intégration de modèles hétérogènes et indépendants, les interfaces proposées restent à un niveau très générique et ne comportent aucun aspect métier lié au domaine visé. En particulier, ces architectures ne proposent pas de stratégie ou de mécanisme permettant la gestion de l'environnement dans lequel le système évolue. De plus, à notre connaissance, il n'est pas prévu de paramétrer une simulation en y introduisant une succession d'événements qui impliquerait plusieurs modules. La notion de scénario de simulation n'est pas nativement intégrée dans ces approches, car elle est souvent fortement liée au domaine d'application.

Deuxième partie

Contributions

Chapitre 6

Les principes généraux d'AMSA

Ce chapitre présente les principes généraux du framework AMSA. Il synthétise les choix qui ont été faits en termes de modélisation, d'architecture et de plateformes ciblées. Une approche détaillée de chacun des points évoqués est présentée dans les chapitres suivants.

AMSA (pour *Advanced Multihull Simulation for Automation*) est un framework qui a pour objectif de faciliter la mise en place de simulateurs pour les contrôleurs de systèmes cyber-physiques, en particulier pour les voiliers de compétition. Le framework est composé d'un ensemble de métamodèles, d'éditeurs et de générateurs de code qui facilite la création d'une architecture pour un simulateur. Un style architectural est proposé afin de gérer les problématiques d'intégration d'éléments hétérogènes. Enfin, une implémentation en C++ d'un *runtime*, ainsi qu'une bibliothèque de composants métier sont également disponibles pour faciliter le développement et l'exécution du simulateur.

6.1 Le modèle à composants dans AMSA

Afin de permettre la construction d'architectures, AMSA propose un modèle à composants adapté au domaine du contrôle et de la simulation. Le modèle définit deux types de composants : des composants atomiques et des composants composites, qui contiennent eux-mêmes des composants. Il est ainsi possible de créer une structure hiérarchique. Les composants atomiques sont instanciés à partir de templates. Ils possèdent des attributs appelés paramètres, initialisés par défaut et qui peuvent être modifiés lors de la simulation. Un template spécifie l'interface du composant via ses entrées et ses sorties, les interfaces pourvues et requises et les paramètres.

Comme ils sont utilisés dans un contexte de simulation, les composants atomiques disposent implicitement de certains attributs qui ne sont pas spécifiés dans les templates : un fréquence d'exécution ainsi que des fonctions d'initialisation, d'exécution de pas de simulation et de finalisation.

A l'inverse des composants atomiques, les composites ne sont pas typés. Ils sont définis uniquement par leurs constituants, ainsi que par des ports d'entrée/sortie, qui sont créés au moment de l'instanciation du composite. Les composants composites sont des singletons, sans comportement, proposés uniquement à des fins de structuration. Il n'y a donc pas de notion de template pour les composites, leur réutilisation est cependant permise par un mécanisme de copie.

Les communications sont orientées donnée, et vu le domaine visé, les données échangées sont numériques de type flottant. Des connecteurs simples relient les ports d'entrée et de sortie des composants. La communication suit un paradigme producteur / consommateur avec écrasement (seule la dernière donnée produite est accessible aux consommateurs). En plus de ces échanges en flux de données (*dataflow*), les composants peuvent fournir ou requérir des interfaces plus complexes. A cet effet, le framework propose deux types d'interfaces : des interfaces de service, destinées à l'échange d'information entre les composants, et des interfaces de marquage, qui permettent de préciser le rôle du composant lors de la simulation.

6.2 La modélisation AMSA

Les différents éléments du modèle à composants AMSA sont décrits et typés via quatre méta-modèles :

Le méta-modèle *AMSA-Interface* contient les éléments nécessaires à la description d'interfaces de communication entre les composants : déclaration de fonctions, définition de types structurés ou énumérés.

Le méta-modèle *AMSA-Template* contient les éléments nécessaires à la description de templates de composants : templates d'entrées/sorties, template de paramètres ainsi que leurs types.

Le méta-modèle *AMSA-Component* contient les éléments nécessaires à la description d'une configuration de composants : les composants atomiques et leurs paramètres, les composants composites, les ports d'entrées/sorties, les ports d'interfaces et les connecteurs.

Le méta-modèle *AMSA-Scenario* contient les éléments nécessaires à la description d'un scénario sur une configuration donnée : principalement des événements (changement de valeur de paramètre ou appel d'interface) et des observateurs (définition et enregistrement des résultats de la simulation).

A partir de ces méta-modèles, trois langages textuels sont proposés pour éditer des modèles d'interfaces, de templates de composants et de scénarios. Les configurations sont créées en instanciant les composants à partir de templates et en les assemblant entre eux à l'aide de connecteurs. Un éditeur graphique permet l'édition des configurations.

6.3 Le style architectural

Les méta-modèles et le modèle à composants présents dans AMSA n'imposent pas, par construction, une architecture particulière pour un simulateur. Cependant, les outils qu'ils fournissent permettent le respect d'un style architectural adapté pour la simulation. Ce style AMSA a un double objectif : proposer une séparation claire des différents constituants des systèmes simulés, et permettre une intégration et une configuration simple de nouveaux composants, afin de gérer l'hétérogénéité des modèles et des systèmes.

Le style AMSA repose ainsi sur deux principes :

L'encapsulation : chaque composant principal de la boucle de contrôle est encapsulé dans un composite. Au premier niveau de description, on obtient ainsi une vision globale de la boucle de contrôle, qui contient les éléments suivants : le système contrôlé (le bateau), le contrôleur (le pilote), l'environnement (le vent), et éventuellement des capteurs ou des actionneurs. A l'intérieur des composites, les composants qui implémentent le comportement des sous-systèmes fournissent une interface de marquage permettant d'identifier leur rôle au sein de la boucle de contrôle.

L'adaptation : Afin de pouvoir intégrer facilement les composants métier, le simulateur s'appuie sur des interfaces et des données standardisées. Le *dataflow* entre les composants du premier niveau respecte un standard communément utilisé dans le monde de la voile, en termes de format de données, de référentiel d'expression et d'unités. Pour intégrer des sources diverses dans le simulateur, il est nécessaire d'avoir recours à des adaptateurs « universels ». Ceux-ci fournissent les entrées et sorties standard du framework, et communiquent avec les composants métier par le biais d'une interface de communication qui intègre diverses possibilités du domaine visé (environnent, bateau, actionneur...). Le composant métier implémente cette interface, l'adaptateur se chargeant de la conversion vers le standard.

De la combinaison de ces deux principes émerge un motif récurrent : chaque sous-système est modélisé par un composite, qui contient lui-même un adaptateur universel pour le sous-système, ainsi qu'un composant métier, relié à l'adaptateur par une interface de communication. Le composant métier fournit également une interface de spécialisation, qui permet d'identifier le type du sous système et d'accéder au composant lors de la conduite de la simulation.

6.4 La génération de code

Une fois l'architecture du simulateur définie, il est possible d'automatiser la production d'une application exécutable grâce aux outils de la génération de code. Par un processus de transformation *model to text*, le framework AMSA permet de générer des classes C++. La génération se déroule en quatre étapes : génération des interfaces, des templates, des composants et des scénarios. Le code généré possède une structure qui traduit le modèle d'architecture.

Génération des templates

L'intégration du comportement et l'éventuel appel au code métier se fait au niveau des templates. La génération d'un template produit un squelette de classe C++ qui contient le prototype des méthodes *doStep()*, *initialize()* et *finalize()*, ainsi que les méthodes de lecture/écriture qui fournissent un accès aux entrées/sorties de la classe. Ces dernières sont des fonctions virtuelles pures (non implémentées dans le template) car leur implémentation dépend de la configuration des composants, qui n'est pas encore connue. Le développeur du composant complète le comportement de celui-ci dans l'implémentation des méthodes, en faisant éventuellement appel à du code ou des bibliothèques existantes.

Génération des composants

Une fois la bibliothèque de templates complétée, les autres phases de génération de code ne nécessitent plus d'intervention externe du développeur. L'édition d'une nouvelle configuration donne lieu à la génération d'autant de classes C++ qu'il y a de composants. Chaque classe de composant hérite de la classe du template auquel est rattaché le composant, et fournit une implémentation aux méthodes de lecture/écriture des entrées et sorties. Cette implémentation dépend de la place du composant dans la configuration. Le comportement et le code métier sont ainsi transmis aux instances de composants par héritage. Les composants atomiques deviennent des classes spécialisées et les interfaces sont converties en classes abstraites. En revanche, les composants composites disparaissent lors de la génération de code, l'ensemble du simulateur est « remis à plat ».

Génération des scénarios

La dernière étape de génération est celle de scénario, elle permet d'obtenir une application exécutable. Une classe en charge de la création, de l'initialisation des composants et de la conduite de la simulation est générée. Le code obtenu a pour seul dépendance la bibliothèque standard C++ et les bibliothèques métier, il peut ainsi être déployé sur une large variété de plateformes.

6.5 AMSA *runtime*

Le framework fournit un ensemble de classes C++ qui permet la création d'un simulateur exécutable à partir du code généré. On y trouve principalement deux ordonnanceurs, dont le rôle est de cadencer l'exécution en appelant les composants de manière cyclique. Une horloge offre le temps simulé et une horloge est en charge du temps réel.

Dans le cas du temps simulé, l'horloge collecte tout d'abord les fréquences d'exécution de chaque composant, elle en déduit leur période d'activation. Le pas de temps unitaire est déterminé en prenant le PGCD de ces périodes. L'horloge détermine ensuite lors de chaque pas quel composant appeler, les méthodes *doStep()* des composants sont toutes appelées à la suite. Un pas de temps commence dès que le précédent est terminé. Cette méthode présente l'avantage d'accélérer considérablement les simulations. Plusieurs heures de navigation peuvent être simulées en quelques secondes.

L'ordonnanceur temps réel fonctionne de manière identique, à ceci près qu'il est synchronisé avec l'horloge du système de la plateforme hôte : un pas de temps ne débute que lorsque l'horloge du système atteint la date de début dudit pas. Ainsi, une heure de navigation requiert une heure de simulation. Cette seconde méthode est utilisée pour permettre à l'utilisateur d'interagir en temps réel avec le simulateur, par exemple pour modifier la consigne du pilote durant la simulation. Elle

est également nécessaire si l'ensemble des composants ne s'exécute pas sur la même machine, ou si le simulateur interagit avec des composants réels (*hardware in the loop*).

Dans tous les cas, l'exécution des composants suit le paradigme *run to completion* : on considère qu'un composant a fini son exécution avant le réveil du composant suivant et avant le prochain pas d'exécution, il n'y a donc pas d'interaction possible lors de l'exécution d'un composant : ses entrées ne varient pas entre le début et la fin de la méthode *doStep()*.

Le framework possède un mécanisme d'ordonnancement basique pour déterminer l'ordre d'appel des méthodes *doStep()* des composants durant un pas de temps : une analyse des dépendances de chaque composant est menée sur le flux de données lors de la génération du code de la configuration, l'ordre d'appel est déduit de la chaîne de dépendance. Si un cycle de dépendances est détecté, l'ordre d'exécution est laissé au choix du créateur de la configuration via la mise en place d'un niveau de priorité associé aux composants.

Chapitre 7

Le framework AMSA

Ce chapitre détaille les principaux mécanismes à la base du framework AMSA : un modèle à composants et sa mise en œuvre pour obtenir un modèle de simulateur. Il aborde ainsi les notions nécessaires pour créer une configuration de composants, puis la mettre en œuvre afin d'obtenir un simulateur. Les bonnes pratiques d'agencement des composants relèvent du style architectural et sont traitées dans le chapitre 8. La manière d'exploiter un simulateur en y jouant divers scénarios fait quant à elle l'objet du chapitre 9.

7.1 Les choix de modélisation

Afin de tirer les bénéfices de l'ingénierie dirigée par les modèles dans la réalisation d'un simulateur, il est nécessaire de modéliser un certain nombre d'éléments. A partir des modèles établis, il est possible, après une étape de vérification, de générer du code afin d'obtenir une application de simulation. La modélisation comprend trois grandes parties : la structure du simulateur, le système de typage utilisé pour décrire cette structure, et les scénarios qui décrivent le déroulement d'une simulation. Certains éléments constitutifs de l'application finale ne sont pas modélisés. C'est par exemple le cas de l'horloge, qui permet de cadencer l'exécution des composants. Une implémentation de ces éléments est cependant fournie avec le framework afin de pouvoir, en les combinant avec le code généré à partir des modèles, obtenir une application complète.

Ce chapitre s'intéresse aux éléments de modélisation au sein du framework AMSA, tant sur les questions de répartition des éléments que sur la stratégie retenue pour la modélisation.

7.1.1 L'approche spécifique

L'objectif du framework AMSA est de faciliter la création de simulateurs de systèmes de pilotage de voiliers compréhensibles et utilisables par les ingénieurs en charge de ces sujets, et dont l'utilisation est aisée pour les navigants et les marins, afin de tirer profit de leurs expériences et de leurs ressentis en termes de pilotage et de navigation. Il est donc important que les concepts manipulés au sein du framework et dans les simulateurs soient les plus proches possible des concepts habituellement rencontrés par les architectes logiciels d'une part, et de ceux utilisés dans le monde de l'électronique marine d'autre part.

Afin d'assurer cette transparence conceptuelle, le choix a été fait de spécialiser au maximum les modèles manipulés, pour qu'ils soient d'eux-mêmes parlants. Une modélisation spécifique de ces concepts a donc été réalisée, sous forme de plusieurs méta-modèles spécifiques à la simulation de systèmes. Ces méta-modèles permettent de décrire les modèles manipulés dans le framework sous forme de langages de modélisation spécifiques (DSML) qui s'approchent au maximum du langage naturel des ingénieurs et des utilisateurs des simulateurs.

Cette approche spécifique permet une meilleure implication des utilisateurs finaux du simulateur dans la phase de conception de celui-ci (au moins en termes de compréhension des phénomènes). Par ailleurs, comme les méta-modèles font partie intégrante du framework, il est relativement simple de les faire évoluer pour y intégrer de nouveaux aspects liés au domaine visé.

7.1.2 La séparation des concepts

La modélisation au sein du framework se fait sur différents plans : les composants du simulateur, les types qui servent à créer ces composants et les scénarios qui permettent de jouer des simulations sur un simulateur existant. La séparation entre ces concepts est clairement marquée par l'emploi de différents méta-modèles. Au sein du framework, un modèle est toujours l'instanciation d'un et d'un seul méta-modèle. Ainsi, les différents concepts sont décrits dans des modèles distincts.

Le framework AMSA comporte quatre méta-modèles. Deux méta-modèles permettent de décrire les concepts de typage (*AMSA-Interface* et *AMSA-Template*), un méta-modèle permet de décrire les composants qui constituent le simulateur (*AMSA-Component*), et un dernier contient les concepts liés aux scénarios de simulation et à l'exploitation des simulateurs (*AMSA-Scenario*). Les méta-modèles sont dépendants entre eux, dans un ordre bien défini : *AMSA-Scenario* dépend de *AMSA-Component*, qui dépend de *AMSA-Template*, qui dépend lui-même d'*AMSA-Interface*. Cette dépendance des méta-modèles entraîne une contrainte lors de l'instanciation, les modèles devant s'enchaîner dans un ordre défini.

Cet ordre de dépendance implique qu'il est tout d'abord nécessaire de déclarer des types avant de pouvoir les utiliser pour créer une structure de composants, et qu'il est ensuite nécessaire de disposer d'une telle structure avant de pouvoir l'utiliser pour jouer des scénarios. D'un point de vue pratique, il est courant de faire des aller-retours entre les modèles lors de la phase de conception, comme cela est expliqué au paragraphe 10.3 traitant des cas d'utilisation du framework. De plus, la séparation de ces concepts dans des modèles différents facilite la réutilisation et la généricité : plusieurs configurations de simulateur peuvent être créées à partir d'un même ensemble de types, et plusieurs scénarios peuvent être appliqués à un même simulateur.

7.2 Le modèle à composants d'AMSA

Le framework repose sur un modèle à composants spécialement adapté aux besoins de notre étude. Chaque composant modélise une entité du système contrôlé, du contrôleur ou de l'environnement. Il s'agit donc avant tout de composants de développement, qui ont pour objectif de structurer le simulateur et donc d'en faciliter la conception.

7.2.1 Structure

Trois objectifs majeurs motivent l'utilisation d'un système à composants : la structuration de l'architecture du logiciel à différents niveaux de granularité, la réutilisation de certaines entités et la gestion de l'hétérogénéité des divers éléments constitutifs d'un simulateur de système cyber-physique. La réponse au besoin de structuration passe par l'utilisation du pattern composite, tandis que la réutilisation des composants est permise par les mécanismes de typage et d'instanciation. La gestion de l'hétérogénéité est gérée via le style architectural d'AMSA, détaillé au chapitre 8.

Le pattern composite

Le modèle à composants d'AMSA respecte le pattern composite où deux types de composants sont présents : les composants atomiques (*LeafComponent*) qui modélisent les briques élémentaires des divers éléments du simulateur, et les composants composites (*CompositeComponent*) qui contiennent d'autres composants. Les composants composites sont des éléments de structuration : ils ne possèdent pas de comportement propre, mais permettent de représenter les systèmes ou les parties des systèmes à divers niveaux de granularité. Cette approche permet d'avoir une vue arborescente du simulateur, dans laquelle il est facile de naviguer pour se placer au niveau de granularité permettant de répondre à une problématique donnée (vision globale pour l'architecte, détail d'implémentation pour le développeur...). La structure de ce système à composants est décrit dans le méta-modèle *AMSA-Core* (figure 7.1).

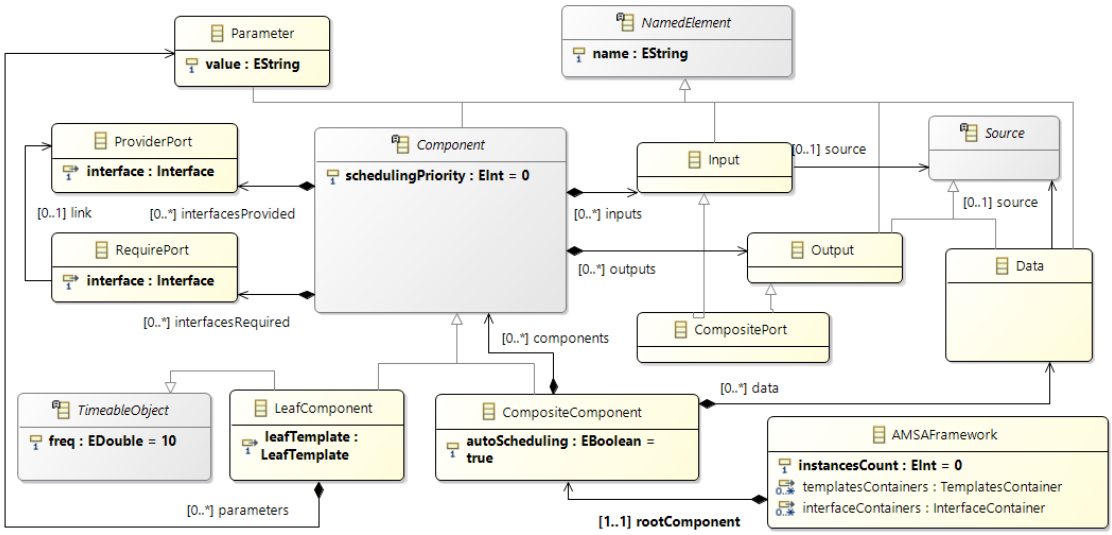

FIGURE 7.1 – Méta-modèle AMSA-Component

Plusieurs composants assemblés entre eux forment une configuration, qui décrit un simulateur. Le point d'entrée de la configuration est un composant composite, appelé composant racine (*Root-Component*). Il contient, de manière directe ou indirecte, tous les composants qui constituent le simulateur. Le composant racine joue un rôle particulier en étant le point d'entrée du modèle du simulateur.

Types et instances

Le mécanisme d'instanciation est différent selon que le composant est atomique ou composite. Les composants atomiques sont instanciés à partir d'un template de composant, qui définit les ports d'entrées/sorties, les interfaces pourvues et requises, ainsi que le nom et le type des paramètres présents dans le composant. La structure de ces templates est définie dans un méta-modèle séparé, *AMSA-Template*, représenté figure 7.2.

Aucun comportement n'est associé a priori à un template car le comportement n'est pas modélisé : il n'est intégré que lors de la mise en œuvre des composants au sein du simulateur en respectant une API standardisée. Ce choix a été fait pour simplifier l'édition de structure du simulateur. De plus, les comportements sont souvent des modèles physiques ou des équations contenus dans du code métier, qui est intégré au simulateur après la génération de code et avant la phase de simulation.

Le comportement d'un template n'a pas d'autres dépendances que les entrées/sorties et les interfaces de celui-ci, ainsi que les propriétés qui lui sont attachées (paramètres et propriétés implicites). Le template contient donc toutes les informations nécessaires à l'implémentation d'un composant. Un composant qui instancie le template peut ainsi utiliser ses propres valeurs de paramètres, les données auxquelles ses entrées/sorties sont connectées, et les autres composants auxquels il est relié par ses interfaces.

L'instanciation de plusieurs composants à partir du même template permet d'obtenir des composants avec un comportement similaire : le même code sera utilisé, les différences dépendront du paramétrage et de la place des composants au sein de la configuration.

Contrairement aux composants atomiques, les composants composites ne sont pas rattachés à un template. En effet, ils n'ont pas de comportement ni de propriété. Un composant composite se résume à une liste de composants contenus, et à une liste de ports d'entrée/sortie. Le type d'un composant composite est donc confondu avec l'instance de celui-ci : chaque composite constitue ainsi un singleton. Le framework ne propose pas de mécanisme pour créer un type de composite sans créer d'instance. Il est cependant possible de dupliquer un composite ainsi que l'ensemble de son contenu (on parle de duplication récursive si plusieurs niveaux de composants sont imbriqués), mais la nouvelle instance obtenue est totalement indépendante de la première.

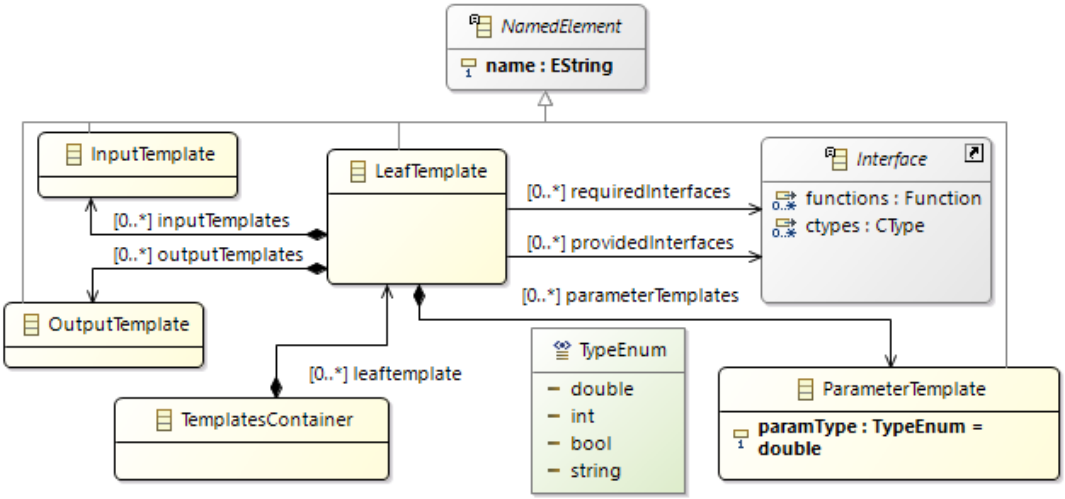

FIGURE 7.2 – Méta-modèle AMSA-Template

Pour gérer l'instanciation des composants, le framework met en place des outils qui permettent :

- d'instancier des composants composites vierges,
- d'instancier des ports d'entrée/sortie composites sur un composant composite,
- d'instancier des composants atomiques à partir d'un template existant. Les ports d'entrée/sortie des composants atomiques ainsi que les ports d'interface sont alors instanciés automatiquement.

Ces outils, intégrés dans les éditeurs de modèles, sont détaillés dans le chapitre 10.

7.2.2 Communication

Le flux de données

Les communications entre les composants se font principalement par échange de données, et suivent le paradigme du *dataflow* à l'image des systèmes embarqués sur les voiliers de compétition. Chaque composant utilise ainsi un certain nombre de données en entrée pour produire de nouvelles données en sortie.

Afin de permettre l'interconnexion des composants, ceux-ci disposent de ports d'entrée/sortie. Chaque port fait transiter une et une seule donnée scalaire. Cela permet d'expliciter l'ensemble des informations qui transitent entre les composants. La transmission d'informations plus complexes est possible via deux mécanismes : soit en utilisant plusieurs canaux (plusieurs ports), par exemple un pour chaque coordonnée d'un vecteur, soit en utilisant la communication par interfaces, dont les mécanismes sont décrits ci-après.

Comme le montre la figure 7.1, un composant composite contient des données (*data*) en plus de contenir des composants. Celles-ci servent à faire le lien entre les ports de sortie et les ports d'entrée des composants : une donnée est reliée à un seul port de sortie, ce qui définit son producteur. En revanche, plusieurs ports d'entrées peuvent être liés à une même donnée, autorisant ainsi de multiples consommateurs. Les données ne sont pas typées dans le méta-modèle : comme de coutume dans l'informatique marine, il s'agit implicitement de valeurs numériques réelles. Les détails de l'implémentation des données (virgule fixe ou flottante, taille de la donnée) sont liés au langage du code généré, et non au modèle du simulateur. Dans le cas de l'implémentation en C++ de la partie *runtime* du framework AMSA, les données seront toujours de type *double*.

Lors de la création d'une configuration, l'édition des liens entre les ports des composants et les données constitue une étape importante, elle permet d'identifier pour chaque donnée son producteur et ses consommateurs. Le flux de donnés entre les composants est alors défini. Cependant, la formalisation de la donnée intermédiaire n'est pas nécessaire dans le modèle, il est également possible de relier directement un port de sortie à un port d'entrée. Les deux méthodes sont équivalentes en termes de code généré.

Les interfaces

Si l'essentiel des communications entre les composants AMSA se fait sous forme de flot de données, des mécanismes d'interaction plus complexes peuvent être nécessaires dans certains cas (échange de méta-données ou synchronisation). Pour répondre à ce besoin, la communication par interfaces est également possible entre les composants. Le concept d'interface est décrit dans un méta-modèle dédié, *AMSA-Interface*, dont une représentation graphique partielle est donnée en figure 7.3. Une interface est composée d'un ensemble de définitions de types complexes (énumérations et structures), et d'un ensemble de prototypes de fonctions. En plus du nom de la fonction, le prototype définit une liste d'arguments, leur type, ainsi qu'un type de retour.

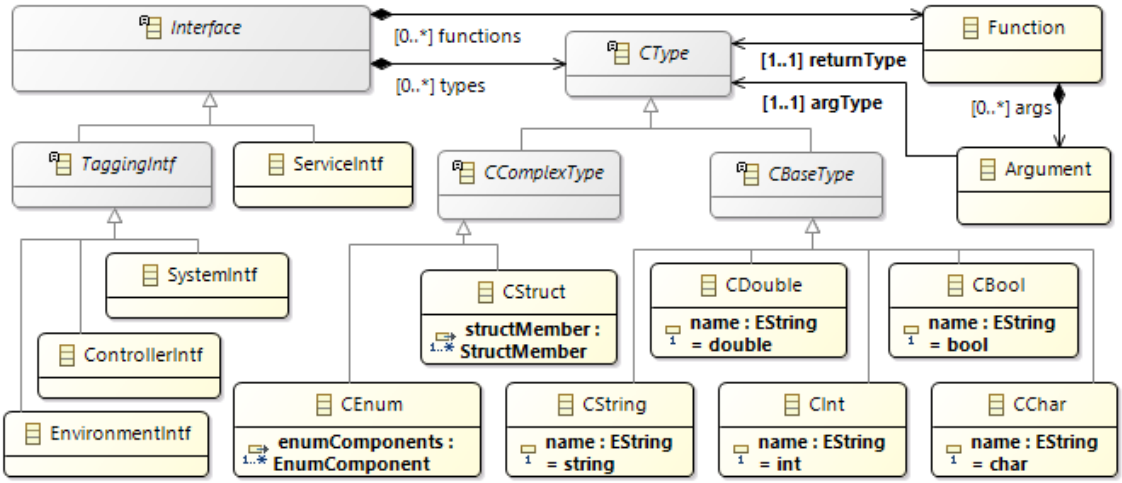

FIGURE 7.3 – Extrait du méta-modèle AMSA-interface

Contrairement aux données relatives aux entrées/sorties, les données manipulées dans les interfaces sont explicitement typées. Le méta-modèle définit les cinq types primitifs communs issus du langage C++ (*int, double, bool, char, string*). Les données manipulées par les fonctions peuvent être soit de type primitif, soit d'un type complexe déclaré dans l'interface.

Pour pouvoir communiquer à travers une interface, les composants disposent de ports d'interface : les ports d'interface fournie (*ProvidedPort*), et les ports d'interface requise (*RequiredPort*) (voir figure 7.1). Chaque port est lié à une interface, une connexion ne peut être établie entre un port fourni et un port requis que s'ils sont liés à la même interface. Un composant qui fournit une interface doit implémenter l'ensemble des fonctions listées dans celle-ci. Un composant requérant une interface peut donc appeler ces fonctions sur le composant auquel il est lié.

Le méta-modèle définit deux catégories d'interfaces : les interfaces de service (*ServiceIntf*) et les interfaces de marquage (*TaggingIntf*). Les interfaces de service sont génériques et sont utilisées entre deux composants pour échanger des informations. Les interfaces de marquage sont spécifiques aux éléments principaux présents dans le simulateur : l'environnement, le contrôleur et le système contrôlé. Elles ne peuvent pas être requises par d'autres composants. Leur rôle est double : identifier les éléments principaux au sein du simulateur (d'où leur nom d'interface de marquage), et permettre l'interaction entre ces éléments et l'extérieur du simulateur (typiquement via un scénario ou un opérateur). Elles sont principalement utilisées dans le cadre du style architectural détaillé au chapitre 8.

Les ports composites

Les mécanismes de communication décrits jusqu'ici permettent l'interaction des composants quand ils sont au même niveau, c'est à dire contenus dans le même composant composite. Une interaction directe entre deux composants n'appartenant pas au même composite est impossible. Cela est nécessaire pour que le composant composite n'ait pas de dépendances autres que celles exprimées par ses propres entrées/sorties.

Pour que les données puissent changer de niveau, c'est à dire rentrer ou sortir d'un composite, ce dernier dispose de ports d'entrées/sorties visibles à la fois de l'extérieur et de l'intérieur du

composite, on parle alors de port composite (*CompositePort*). Un port composite d'entrée sera perçu comme un port d'entrée à l'extérieur du composite, et comme un port de sortie à l'intérieur du composite, car il permet de fournir une donnée pour les composants internes. A l'inverse, un port composite de sortie sera perçu comme un port d'entrée à l'intérieur du composite, car il attend une donnée interne pour pouvoir la proposer aux composants extérieurs.

Les ports composites agissent ainsi comme des tunnels qui permettent aux données de traverser la paroi du composant composite. Ils permettent également de créer une indirection sur les données lors du passage de cette paroi, ce qui permet de modifier la configuration interne du composite indépendamment du contexte externe dans lequel il est utilisé, tant que la nature des entrées et sorties ne change pas.

7.2.3 Propriétés

Les composants peuvent disposer d'un certain nombre de propriétés. Certaines sont génériques, dans le sens où elles sont automatiquement présentes dans tous les composants. De ce fait, elles ne sont pas déclarées dans le template du composant. Il est possible d'ajouter des propriétés additionnelles aux composants en les déclarant dans leur template. Comme les composants composites ne sont pas rattachés à un template, ils ne disposent que de propriétés implicites.

Les propriétés génériques

Tous les composants disposent d'un nom, qui permet de les identifier, et également d'une propriété *schedulingPriority*, de type entier, qui permet de déterminer un ordre d'appel (l'utilité et les mécanismes de cet ordre sont discutés en 7.3.3). Les composants composites disposent en plus d'une propriété de type booléen, *autoScheduling*, qui indique si l'ordre d'appel de leurs enfants doit être déterminé automatiquement. Enfin, les composants atomiques disposent d'une propriété *frequency*, de type *double*, qui détermine la fréquence à laquelle ils seront exécutés. Toutes ces propriétés sont définies dans le méta-modèle *AMSA-Component* (voir figure 7.1).

Les paramètres

Le méta-modèle *AMSA-Template* (figure 7.2) permet la déclaration de paramètres dans un template. Un paramètre est caractérisé par un nom et un type. Quatre types primitifs sont disponibles dans le méta-modèle : *integer, double, boolean* et *string*. Lorsqu'un paramètre est déclaré dans un template, tous les composants rattachés à ce template disposent de ce paramètre, mais chacun peut disposer d'une valeur différente : la valeur du paramètre est contenue dans un champ *value*, qui est propre au composant, et non au template. Ce système de paramétrage permet une analogie avec le paradigme Orienté Objet : il existe ici le même type de relation entre un composant et son template que celle que l'on trouve entre un objet et sa classe.

Les contraintes

Un jeu de contraintes est mis en place dans le framework afin de s'assurer du bon usage des propriétés. Ces contraintes font partie intégrante des méta-modèles. Elles sont formalisées dans un langage dédié : le langage *OCL* [OMG14]. En particulier, une contrainte sur les noms des composants implique que tous les enfants d'un même composant composite doivent avoir un nom différent. Ainsi, en utilisant le nom du composant préfixé récursivement du nom de ses parents, on obtient un identifiant unique pour chaque composant. Une autre contrainte impose que la fréquence d'un composant soit strictement positive.

Le mécanisme des contraintes est également utilisé pour vérifier que la valeur d'un paramètre est bien conforme au type déclaré dans le template : pour des raisons de simplicité dans les éditeurs, la valeur est stockée sous forme d'une chaîne de caractères, les contraintes vérifient que cette chaîne est convertible dans le type requis.

7.2.4 Comportement

La notion de comportement est principalement liée aux composants atomiques. Le comportement des composants composites est implicitement défini comme la composition des comportements de leurs enfants. Pour les composants atomiques, le comportement est lié non pas à l'instance du composant, mais au template, à l'image des modèles orientés objet : l'instance utilise le comportement défini au niveau du template.

Le comportement des composants atomiques n'est pas décrit par le modèle, mais par ajout de code métier. Cependant, ces composants disposent d'une fréquence d'exécution, ce qui définit implicitement une activité interne : chaque composant va exécuter son activité de manière périodique. Cette activité va permettre d'implanter un comportement actif dans les composants, soit l'exécution périodique d'une méthode *doStep()*. Les composants peuvent également disposer d'un comportement passif s'ils fournissent une ou plusieurs interfaces : l'appel aux fonctions de ces interfaces va déclencher une exécution interne.

L'intégration du comportement dans les composants se fait usuellement en appelant des classes ou des bibliothèques qui respectent des interfaces bien définies. Le framework AMSA propose de générer le squelette de ces classes à partir des templates, afin de faciliter leur écriture et de permettre l'intégration de code existant. Les mécanismes de génération sont détaillés dans le chapitre 10.

7.3 Mise en œuvre pour la simulation

Après avoir détaillé la structure du modèle à composants d'AMSA, cette partie s'intéresse à la manière de mettre en œuvre une configuration de composants de développement pour obtenir un simulateur de pilotage de voilier. En effet, même si le modèle à composants d'AMSA est avant tout un modèle de développement, il est également utilisé pour générer du code afin d'obtenir un simulateur exécutable.

Le générateur de code fourni avec le framework AMSA permet de générer des classes C++ à partir d'une configuration de composants, des interfaces, des templates et des scénarios qui y sont liés. Les instances de ces classes sont alors considérées comme des composants logiciels. Cette partie s'intéresse à la mise en œuvre des composants ainsi obtenus et à la manière de les exécuter afin de simuler le pilotage d'un voilier.

7.3.1 Le cycle de vie des composants

Les composants logiciels disposent de trois méthodes permettant de gérer leur cycle de vie à l'exécution :

- la méthode d'initialisation (*initialize()*) prépare le composant à l'exécution. Elle permet l'initialisation des paramètres du composant, la découverte des composants auxquels il est lié par une interface, ainsi que toute autre action d'initialisation nécessaire à l'exécution du comportement (réservation de ressources, ouverture de fichier...) ;
- la méthode d'exécution (*doStep()*) est appelée périodiquement durant l'exécution.
- la méthode de finalisation (*finalize()*) est appelée à la fin de l'exécution du composant, elle est en charge de libérer les ressources utilisées et éventuellement de sauvegarder certains résultats ;

Ces méthodes ne sont pas présentes dans les modèles, car leur prototype est invariant et commun à tous les composants. Leur implémentation relève du code métier.

7.3.2 La conduite d'une simulation

Une fois que les composants sont initialisés, la conduite d'une simulation passe par un appel régulier à la méthode *doStep()*. Cet appel doit être fait à la fréquence indiquée par le paramètre *frequency* du composant. Les appels sont faits à partir d'une horloge centralisée. L'horloge détermine, selon un pas de temps élémentaire compatible avec l'ensemble des fréquences, quels composants appeler à chaque pas de temps. Les composants alors sont appelés de manière séquentielle.

L'appel des composants par l'horloge respecte le paradigme du *run to completion*, ce qui implique qu'il n'y a ni interruption, ni interaction avec d'autres composants lors de l'exécution d'une méthode *doStep*. Lors d'un appel, le composant lit les données d'entrée, il exécute son comportement *doStep()*, puis écrit sur ses ports de sortie. Le composant suivant n'est appelé que lorsque le composant précédent a fini son exécution, il n'y a pas à ce jour d'exécution parallèle.

Pour assurer les communications, les composants mettent à jour les données qu'ils publient en sortie à chaque exécution de leur méthode *doStep()*. Lors de leur propre exécution, les autres composants viennent lire ces valeurs.

7.3.3 L'ordonnancement des composants

Lorsque l'horloge doit exécuter plusieurs composants, il est nécessaire de déterminer un ordre d'appel car les appels sont séquentiels. Une analyse du flux de données est menée dans chaque composant composite dont la propriété *autoScheduling* est vraie, dans l'objectif que l'ordre d'appel respecte l'ordre du flux : un composant qui produit une donnée en sortie doit être appelé avant un composant qui utilise cette même donnée en entrée.

Les entrées du composite ainsi que les sorties des enfants ne possédant pas d'entrée sont considérées comme données initiales, les sorties du composite sont considérées comme données finales. Les enfants sont ordonnés pour respecter l'ordre du flux. Si une boucle de dépendance est détectée (ou si la propriété *autoScheduling* du composite est fausse), l'algorithme utilise les valeurs des propriétés *schedulingPriority* de chaque enfant pour déterminer l'ordre d'appel : les plus basses seront appelées en premier. En cas d'égalité, l'ordre de création des composants dans le modèle est utilisé.

Une fois cette analyse faite pour chaque composant composite, un ordre global est déterminé pour les composants atomiques (rappelons que les composites disparaissent lors de la génération), en partant du *rootComponant*, et en remplaçant récursivement les composites par l'ensemble de leurs enfants.

7.3.4 La simulation temps réel ou temps simulé

Au niveau de la simulation de l'écoulement du temps, deux approches sont possibles, selon le cas d'utilisation envisagé pour le simulateur : soit la simulation est effectuée en temps réel (la simulation d'un pas de temps suit une horloge temps réel), soit la simulation est conduite en temps simulé : le calcul d'un pas de temps est lancé immédiatement après la fin du calcul du pas précédent.

La seconde approche permet de gagner du temps de simulation de manière substantielle (selon la lourdeur des modèles utilisés, plusieurs heures de navigation peuvent être simulées en quelques secondes ou minutes). Cependant, elle n'est pas applicable dans certaines conditions :

- lorsque la simulation requiert une intervention de l'utilisateur, par exemple pour ajuster manuellement la consigne du pilote ou son paramétrage ;
- lorsque certains composants du simulateur s'exécutent sur une plateforme différente (calculateur séparé) ou en dehors du framework (exécutable séparé, non synchronisé avec le simulateur) ;
- lorsque des éléments physiques (capteurs, actionneurs) interviennent dans la simulation (situation dite *hardware in the loop*), leur réponse dynamique impose une simulation temps réel.

Le framework permet de configurer l'horloge pour prendre en compte les deux approches. Dans tous les cas, l'horloge respecte les principes énoncés précédemment (*run to completion* et appels séquentiels). Dans le cas du temps réel, il est évidemment impératif que l'hypothèse synchrone soit respectée : chaque appel doit finir de s'exécuter avant le début du suivant.

Étant donnée la relative légèreté des modèles utilisés actuellement pour la simulation de pilotage, cette hypothèse est a priori toujours valide et le framework ne prévoit donc pas de mécanisme pour la garantir. Cependant, un message d'avertissement est affiché au cours de l'exécution si elle vient à ne pas être respectée.

Chapitre 8

Le style architectural d'AMSA

Le modèle à composants du framework AMSA offre des outils pour définir et structurer des modèles d'application de simulation. Cependant, la manière d'agencer les composants entre eux n'est pas imposée par les modèles : cela relève d'un ensemble de bonnes pratiques et d'usages liés au domaine d'application. L'ensemble de ces bonnes pratiques et de ces motifs récurrents forme un style architectural. Ce chapitre décrit les principaux aspects d'un style architectural adapté pour utiliser le framework AMSA à des fins de simulation pour les voiliers et leurs pilotes.

Le style architectural d'AMSA permet, en s'appuyant sur la modélisation et les possibilités offertes par le modèle à composants, de résoudre deux problèmes qui se posent lors de la conception d'une application dédiée à la simulation de pilotage de voiliers : d'une part la vision multi-échelles du système, qui permet d'aborder la complexité à divers niveaux de granularité, et d'autre part la gestion de l'hétérogénéité et de l'intégration des divers éléments constitutifs du simulateur.

8.1 Vision globale du simulateur

Les composants composites permettent d'aborder le simulateur modélisé à plusieurs échelles : le *rootComponent* offre une vision globale des principaux constituants du simulateur et du flux de données échangées. Chaque constituant est lui-même un composite, ce qui permet de modéliser son organisation sous une forme hiérarchique. Plusieurs niveaux peuvent ainsi être enchaînés, jusqu'à obtenir la précision souhaitée pour la modélisation des composants élémentaires.

8.1.1 La boucle de contrôle

Comme le système simulé est un système de contrôle, le premier niveau est constitué des éléments standards d'une boucle de contrôle. On y trouve le système contrôlé, le contrôleur, les capteurs et les actionneurs. La figure 8.1 synthétise les éléments constitutifs de cette boucle de contrôle, ainsi que les principaux flux de données qui transitent entre ces éléments. Comme un voilier est un système cyber-physique dont le comportement dépend fortement de l'environnement dans lequel il évolue, l'inclusion d'un modèle de l'environnement dans la boucle de contrôle est aussi nécessaire.

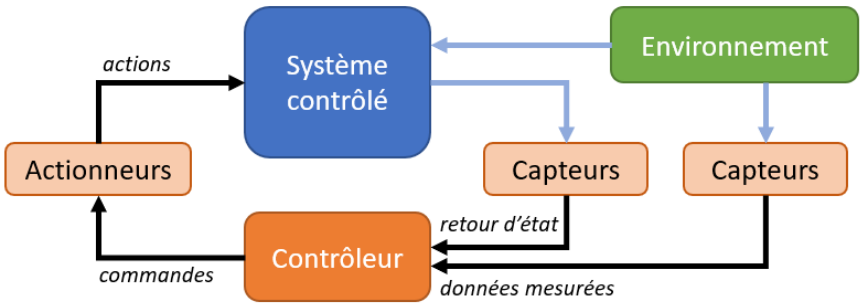

FIGURE 8.1 – Les principaux éléments de la boucle de contrôle et leurs interactions

Si l'environnement agit sur le système contrôlé, il prend également part au processus de régulation. Ainsi le contrôleur obtient des informations en provenance à la fois du système qu'il contrôle et de l'environnement. C'est la raison pour laquelle deux capteurs apparaissent sur la figure 8.1.

La figure 8.2 montre une boucle de contrôle typique modélisée dans le framework AMSA, il s'agit d'une capture d'écran de l'éditeur graphique de modèles d'*AMSA-Component*, montrant le contenu d'un *rootComponent*. On y retrouve les éléments principaux de la boucle de contrôle : le bateau (*Boat*) en tant que système contrôlé, le pilote (*Pilot*) comme contrôleur. On note que les capteurs ne sont pas modélisés ici, pour des raisons exposées en 8.1.2.

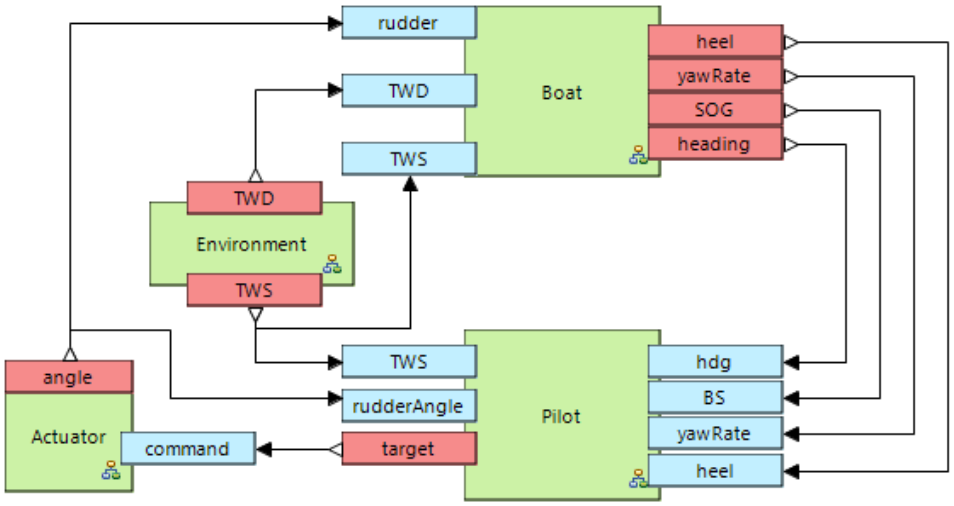

FIGURE 8.2 – Le contenu d'un *rootComponent* classique. Les rectangles verts représentent des composants composites, leurs sorties sont bleues et leurs entrées rouges.

Puisque la boucle de contrôle contient, à peu de choses près, toujours les mêmes éléments, il paraît intéressant de pouvoir identifier ces éléments par leur nature. Cela permet par la suite d'automatiser des opérations de vérification sur les modèles, en s'assurant de la cohérence de la boucle de contrôle. L'identification de la nature des composants est également utile lors de la création de scénarios, comme expliqué dans le chapitre 9.

Comme un des objectifs du framework est d'être évolutif et adaptable à diverses situations (intégration de nouveaux modèles de bateaux, réutilisation pour un métier différent...), l'architecture du *rootComponent* et la présence des éléments standards de la boucle de contrôle ne sont pas figées, et c'est le mécanisme des interfaces de marquage qui permet d'identifier le rôle des composants. En effet, le méta-modèle *AMSA-Interface* propose trois types d'interfaces de marquage, qui correspondent aux trois principaux constituants de la boucle de contrôle (voir figure 7.3) : le type *SystemIntf* pour le marquage des composants qui modélisent le système contrôlé, *ControllerIntf* pour ceux qui modélisent le contrôleur, et *EnvironmentIntf* pour ceux qui modélisent l'environnement.

Un composant qui fournit une interface de marquage est étiqueté comme ayant un rôle particulier dans la boucle de contrôle. Cet étiquetage n'est pas obligatoire dans la modélisation AMSA, il relève purement du style architectural. Le comportement du composant n'est pas modifié par l'étiquette, mais le composant doit implémenter les méthodes définies par l'interface. Les interfaces de marquage, à l'inverse des interfaces de service, ne peuvent pas être requises par d'autres composants. Elles servent à manipuler le composant depuis l'extérieur de la configuration, soit par le biais d'une interface homme / machine (interaction directe avec l'utilisateur), soit au sein d'un scénario. L'étiquetage prend ici tout son sens, car il permet d'identifier le « type » de composant manipulé.

Comme les interfaces ne peuvent être fournies que par les composants atomiques, ce ne sont pas les composites contenus dans le *rootComponent* qui portent les interfaces de marquage, mais leurs enfants. Pour conclure, selon le style architectural AMSA, la vision globale du simulateur est constituée d'un *rootComponent* qui contient lui-même des composants composites, un par grand constituant de la boucle de contrôle (système, contrôleur, environnement, et éventuellement capteurs et actionneurs). Chaque composite contient lui-même des composants, dont le rôle de

certains est étiqueté par une interface de marquage.

8.1.2 Discussion sur les capteurs et actionneurs

Une boucle de contrôle réelle nécessite au moins un actionneur (qui transforme la consigne du contrôleur en action physique sur le système contrôlé), et au moins un capteur si le contrôle est fait en boucle fermée, son rôle étant de fournir au contrôleur des données sur l'état du système. Si l'on souhaite simuler un système qui contient une boucle de contrôle, il paraît donc naturel de modéliser capteurs et actionneurs, afin de simuler l'impact qu'ils ont sur le contrôle. En effet, les capteurs peuvent engendrer des erreurs, du bruit et des retards sur les données mesurées, tandis que les actionneurs induisent un temps de réponse entre la consigne et l'action, et possèdent souvent une réponse dynamique qui leur est propre.

Cependant, la modélisation des erreurs et retards (que l'on nomme par la suite altérations) induits par les capteurs implique deux choses : d'une part il est nécessaire de connaître la nature et les caractéristiques des altérations, caractéristiques qui sont propres au capteur et à l'environnement dans lequel il évolue, et d'autre part il faut pouvoir disposer dans le simulateur de la donnée non altérée pour pouvoir lui appliquer l'altération. Si des modèles de bruits et de retards existent, il est impossible dans certains cas de connaître l'évolution de la donnée non altérée.

Les modèles physiques utilisés dans notre étude reposent souvent sur des données acquises en conditions réelles (donc par le biais de capteurs) : elles peuvent servir à identifier divers paramètres du modèle, ou peuvent constituer le modèle lui-même (c'est le cas par exemple d'un enregistrement de vent, qui peut être utilisé comme composant d'environnement). Ces modèles fournissent des données issues des capteurs et intègrent donc des données bruitées. Dans ce cas, le modèle physique modélise un système et ses capteurs de manière indissociable. A contrario, certains modèles purement théoriques produisent des données non bruitées, qu'il est intéressant de faire transiter par des modèles de capteurs. Deux solutions sont envisageables pour gérer ces diverses situations :

- modéliser les capteurs seulement lorsque cela est nécessaire. La souplesse du modèle à composants permet d'insérer et d'enlever facilement des composants au sein du *dataflow*.
- toujours modéliser les capteurs, mais faire en sorte que ceux-ci n'aient pas d'effet lorsqu'un modèle produisant des données déjà altérées est utilisé. Le modèle du capteur se résume alors à l'identité, il recopie ses entrées sur ses sorties.

Le framework AMSA n'impose pas de choix par rapport à cette problématique. Les deux solutions peuvent être adaptées en fonction des types de modèles manipulés. Cependant, l'ensemble des modélisations décrites dans ce document font principalement intervenir des modèles basés sur des données mesurées. La première solution a donc été retenue et, dans ce qui suit, les capteurs ne sont modélisés que lorsque cela est nécessaire.

L'approche est différente pour ce qui concerne les actionneurs : en effet, sur les systèmes réels, il est souvent possible d'obtenir de manière indépendante la consigne demandée à l'actionneur et l'action qui en résulte sur le système (au biais du capteur près). Il est donc plus simple de modéliser le comportement d'un actionneur indépendamment du système qu'il actionne, ou du contrôleur qui le commande, même lorsque le comportement de ceux-ci est identifié à partir de données mesurées. De plus, l'objectif des simulations est souvent d'étudier le contrôleur indépendamment de l'actionneur. Comme celui-ci fournit une consigne et non une action, la modélisation de l'actionneur apparaît comme inévitable.

8.2 La gestion de l'hétérogénéité

Outre la possibilité de modélisation du système à plusieurs échelles, le style architectural AMSA permet de répondre à un problème commun à la simulation et aux systèmes cyber-physiques : la grande hétérogénéité des sources et des modèles employés au sein d'un même simulateur. Pour cela, le style architectural préconise l'emploi d'un motif récurrent, dit *pattern d'adaptation*, qui fait intervenir un adaptateur utilisant des interfaces standardisées propres au domaine d'application.

8.2.1 Les sources d'hétérogénéité

Comme abordé dans la partie précédente, la simulation d'un système de contrôle de voilier requiert l'usage de divers modèles, tant pour le bateau que pour l'environnement. Ces modèles, ainsi que les lois de contrôle employées, peuvent prendre des formes très diverses. Il en émerge trois grandes sources d'hétérogénéité :

le format des données présentes en entrée et sortie des modèles : elles sont caractérisées par leur type (scalaires, vectorielles, continues ou discrètes), leur unité, leur référentiel d'expression.

la richesse des modèles qui impacte le nombre d'entrées et de sorties. Certains modèles de contrôleurs peuvent être très simples et ne considérer que quelques éléments comme la vitesse et l'orientation du bateau, tandis que d'autres modèles de contrôleurs peuvent travailler avec un ensemble de données beaucoup plus complexes telles que les données inertielles ou l'attitude de la plateforme.

la nature des modèles : certains modèles sont purement théoriques, d'autres sont identifiés à partir de situations réelles. Il doit être aussi possible d'inclure une partie du système physique dans la boucle de simulation (on parle alors de simulation *hardware in the loop*). Les données manipulées par les modèles peuvent ainsi être de natures diverses (données théoriques, issues d'un modèle physique, d'un enregistrement, d'un système réel...).

L'objectif du pattern d'adaptation est de pouvoir utiliser des modèles très différents au sein d'un même simulateur, et donc de les interfacer entre eux sans passer par une phase de configuration et d'adaptation fastidieuse. Cette facilité d'adaptation permet par la suite d'interchanger les modèles à loisir en fonction des besoins de la simulation. Elle facilite également l'introduction de nouveaux types de modèles dans le simulateur.

8.2.2 Le pattern d'adaptation

Le pattern d'adaptation repose sur le principe d'un adaptateur universel, qui est en charge d'adapter son interface de communication afin de la faire correspondre à des entrées et des sorties standardisées au sein du framework.

Pour réaliser une telle adaptation, l'adaptateur universel doit connaître les caractéristiques du modèle qu'il adapte : quelles sont les données produites, quelles sont les données nécessaires, quelles sont les unités et conventions employées, quelle est la fréquence de production des données. Il est donc nécessaire qu'une négociation ait lieu entre le modèle à intégrer et l'adaptateur lors de la phase d'initialisation, afin de fixer ces paramètres.

Cette négociation s'effectue à travers une interface de service, que l'on nomme interface d'adaptation, qui est fournie par le composant contenant le modèle et requise par l'adaptateur. Selon la nature du modèle à adapter, cette interface contient des méthodes et des types permettant la négociation sur les unités, les référentiels et la nature des données. Lors de la simulation, comme l'échange de données qui a lieu entre le modèle et l'adaptateur dépend du résultat de la négociation (en termes de nombre de données, de leur nature et de leur type), il ne peut pas se faire par les mécanismes classiques du *dataflow*. L'interface d'adaptation offre donc des méthodes pour permettre l'échange des données au cours de la simulation.

Les données d'entrée et de sortie de l'adaptateur étant standardisées, le mécanisme de *dataflow* est utilisé pour connecter ce dernier aux autres composants du simulateur. Cependant, dans la boucle de simulation, le pattern d'adaptation prévoit que l'adaptateur et le modèle auquel il est connecté soient encapsulés dans un composant composite, dont les ports d'entrée/sorties correspondent à celui de l'adaptateur. Cela est visible sur la figure 8.3 qui présente l'intérieur d'un composite suivant le pattern d'adaptation. Les entrées et sorties de l'adaptateur *windAdaptor* (respectivement *lat, lon* et *TWS, TWD*) coïncident avec les entrées et sorties du composant composite.

Lorsque l'on dispose d'un adaptateur, l'utilisation d'un modèle dans le simulateur nécessite l'écriture d'un *wrapper* de ce modèle pour implémenter l'interface d'adaptation. Ce *wrapper* se présente sous la forme d'un composant que l'on inclut dans la configuration du simulateur, en le reliant à l'adaptateur à l'aide de l'interface d'adaptation. La figure 8.3 présente par exemple trois *wrappers* pour trois modèles de vent différents (*WindGenerator*, *ConstantWind* et *FileWindReader*). Les trois implémentent l'interface d'adaptation *windAdaptation*, mais seul le troisième est lié à l'adaptateur par un lien d'interface. L'édition de ce lien permet de sélectionner quel modèle est utilisé au sein du simulateur. Comme le pattern d'adaptation est utilisé pour les éléments de la

boucle de contrôle, il est fréquent que les *wrappers* fournissent également une interface de marquage permettant d'identifier leur rôle dans le simulateur.

En conclusion, le pattern d'adaptation est composé des éléments suivants :

- Un composant composite dont les entrées et sorties sont standardisées : il s'interconnecte avec les autres composants du simulateur, et contient les autres éléments du pattern ;
- Un adaptateur universel, dont les entrées / sorties coïncident avec celles du composant composite. Il requiert une interface de service qui lui permet de communiquer avec le système à adapter ;
- Un ou plusieurs composants atomiques qui contiennent le ou les modèles hétérogènes (soit directement, soit sous forme de *wrapper*). Chaque composant fournit l'interface de service évoquée précédemment afin de pouvoir communiquer avec l'adaptateur. Il peut également fournir une interface de marquage qui permet de définir son rôle dans le simulateur et de le manipuler de l'extérieur durant une simulation.
- Un connecteur d'interfaces qui relie l'adaptateur à un composant atomique : il permet de faire la sélection du modèle à utiliser.

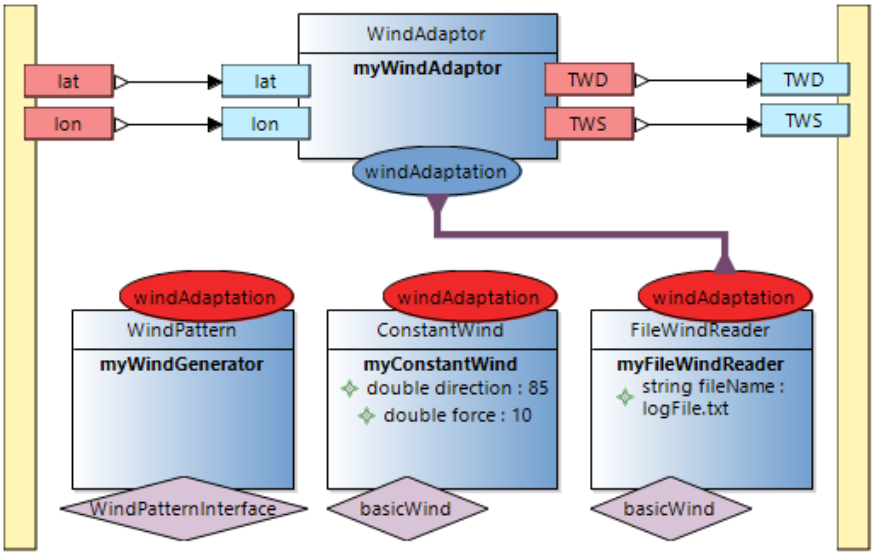

FIGURE 8.3 – Un exemple de composant composite respectant le pattern d'adaptation. Les bandes jaunes modélisent les frontières du composite, les rectangles bleus sont des composants atomiques. Les interfaces de service sont représentées sous forme d'ellipses, et les interfaces de marquage sous forme de losanges.

Le respect de ce pattern d'adaptation permet de gérer l'hétérogénéité des modèles employés pour la simulation, via la sélection du modèle désiré par simple édition d'un connecteur d'interfaces au sein de la configuration de composants. Cependant, pour pouvoir être applicable, le pattern requiert que des interfaces standards aient été définies, tant pour la communication entre modèle et adaptateur que pour les entrées et sorties de l'adaptateur. Ces interfaces sont fortement liées au domaine concerné et aux pratiques « métier ».

8.2.3 L'adaptation des modèles de vent

Le vent est l'élément d'environnement principal qui influence le voilier. L'état de mer n'est pas pris en compte dans cette étude. Dans ce qui suit, l'environnement du système se résume donc au vent.

Les entrées / sorties standard

Il est tout d'abord nécessaire de définir les entrées et sorties d'un adaptateur universel pour les sources de vent. Dans le cadre de la simulation, on s'intéresse exclusivement au vent reçu par le bateau. Le vent est représenté comme un vecteur, ou comme un champ de vecteurs en deux

Entrées			Sorties		
Grandeur	**Unité**	**Nombre de données**	**Grandeur**	**Unité**	**Nombre de données**
Position	degré	2	Vitesse du vent	nœud	1
			Direction du vent	degré	1

TABLE 8.1 – Entrées et sorties standard des modèles de vent

dimensions. L'adaptateur fournit en sorties les deux coordonnées du vecteur vent. Pour ce faire, il a besoin en entrée des deux coordonnées correspondant à la position du bateau, position à laquelle on souhaite obtenir le vent.

Il est nécessaire de standardiser le format de ces données au sein du framework afin que les composants puissent les utiliser. On choisit ici les conventions les plus répandues dans le domaine de la voile : le vent est exprimé sous forme d'une vitesse, exprimée en nœuds, et d'une direction, exprimée en degrés par rapport au nord géographique : un vent du Nord a pour direction 0°, alors qu'un vent d'Est a pour direction 90°. Ces coordonnées de vent, exprimées dans le référentiel terrestre, sont souvent abrégées en TWS et TWD, respectivement pour *True Wind Speed* et *True Wind Direction*. Les positions géographiques sont quant à elles exprimées en latitude et longitude dans le système WGS84 (communément nommées coordonnées GPS). Les coordonnées sont exprimées en degrés décimaux. Le tableau 8.1 récapitule les entrées et sorties standardisées pour les modèles de vent.

Notons que l'information temporelle, qui peut intervenir dans la définition du vent (si celui-ci n'est pas constant), n'est pas prise en compte dans les entrées. En effet, le temps n'est pas modélisé en tant que donnée dans le modèle à composants. Il intervient directement au sein des composants, via l'horloge à laquelle ils sont reliés.

On fait ici l'hypothèse que le vent est dans un plan horizontal, et constant verticalement, donc que l'ensemble du bateau reçoit le même vent à un instant donné (le vecteur vent a seulement deux coordonnées, les points géographiques également). Cette hypothèse est raisonnable par rapport aux modèles utilisés pour la simulation. Cependant, une prise en compte d'une dimension verticale ne nécessiterait qu'une redéfinition partielle de l'adaptation.

L'interface *windAdaptation*

L'interface *windAdaptation*, qui assure la communication entre l'adaptateur et le modèle fournissant le vent, a un double rôle. Elle doit permettre une négociation lors de l'initialisation sous forme d'échange de métadonnées afin que l'adaptateur puisse s'adapter aux caractéristiques de la source. Elle doit aussi permettre la transmission des données durant la simulation.

Comme les données concernées sont sous forme de vecteur ou de champ de vecteurs non constants dans le temps, l'interface prévoit les différentes possibilités pour exprimer de telles données. Ainsi, lors de la négociation initiale, la source doit indiquer :

une stratégie temporelle : soit les données sont produites à fréquence fixe (*synchronous*), soit il est nécessaire de préciser l'instant auquel on souhaite obtenir les données (*asynchronous*).

une stratégie spatiale : soit la source produit un vecteur unique indépendant de la position (*nonLocalized*), soit elle produit un champ, et dans ce cas, la position à laquelle on souhaite les données doit être renseignée (*localized*).

un référentiel d'expression des données : il peut être cartésien ou polaire, auquel cas il est nécessaire de préciser la direction d'origine et le sens de rotation.

les unités utilisées pour exprimer les données : unités de vitesse, de longueur et unités angulaires.

En supplément, la source fournit également deux métadonnées qui renseignent sur la nature des données transmises : l'origine des données (capteur, enregistrement, modèle physique, prévision météo), ainsi que la hauteur à laquelle le vent est fourni (on considère souvent que les marins raisonnent sur le vent à 10 mètres de hauteur car il correspond au vent qu'ils lisent sur le plan d'eau). Ces informations peuvent être utiles à l'adaptateur s'il souhaite par exemple ajouter du bruit ou un retard dus au capteur sur un vent théorique, ou encore « redescendre » à 10 mètres le vent qui aurait été mesuré sur un point haut (plusieurs modèles de profil vertical de vent sont utilisables pour une telle opération).

L'interface *windAdaptation* définit un type structuré qui contient un champ pour chacune de ces informations. De même, des énumérations sont définies pour les métadonnées non scalaires. Ainsi, à chaque source de vent correspond une variable structurée, en quelque sorte sa signature. Cette variable, dont on peut voir un exemple de déclaration en C++ sur la figure 8.4, est transmise à l'adaptateur lors de l'initialisation.

```
windAdaptation::InitValues metadata;
metadata.timeStrategyValue = windAdaptation::synchronous;
metadata.spaceStrategyValue = windAdaptation::nonLocalized;
metadata.spaceCoordinatesStrategyValue = windAdaptation::polar;
metadata.timeUnitValue = windAdaptation::hz;
metadata.dateUnitValue = windAdaptation::timeStamp;
metadata.zUnitValue = windAdaptation::m;
metadata.speedUnitValue = windAdaptation::kts;
metadata.angleUnitValue = windAdaptation::deg;
metadata.zeroDirectionValue = windAdaptation::north;
metadata.sensOfRotationValue = windAdaptation::antiTrigo;
metadata.timeOrFrequencyValue = 10;
metadata.zAltitudeValue = 10;
metadata.dataOriginValue = windAdaptation::model;
```

FIGURE 8.4 – Exemple de déclaration des métadonnées liées à une source de vent, en langage C++

En fonction des stratégies temporelles et spatiales négociées à l'initialisation, la communication des données se fait de manière différente : si la source est synchrone et non localisée, l'adaptateur se contente d'interroger celle-ci à fréquence fixe pour obtenir des données. Si la source est asynchrone, l'adaptateur précise l'instant auquel il veut les données à chaque pas d'exécution. Si la source est localisée, l'adaptateur précise en plus les coordonnées du lieu à chaque demande. L'interface définit donc différentes fonctions pour la communication, avec des prototypes adaptés à chaque situation. La mise en œuvre de cette interface au sein d'une configuration est illustrée sur la figure 8.3.

8.2.4 L'adaptation des modèles de bateaux

Les entrées / sorties standard

De même que pour les modèles de vent, il est nécessaire d'établir une liste exhaustive des entrées et des sorties qui sont prises en compte par l'adaptateur des modèles de bateaux. Cependant, dans le cas d'un modèle de bateau, le nombre et la richesse des données présentes en sortie peuvent varier fortement en fonction de la complexité des modèles utilisés. Il est donc nécessaire de prendre en compte les multiples possibilités susceptibles d'être rencontrées dans un modèle de simulation, sans pour autant trop augmenter la complexité de l'adaptateur.

La stratégie retenue consiste à pourvoir l'adaptateur des entrées et sorties susceptibles d'être utilisées dans le cadre de simulation pour le pilotage. Lors de l'utilisation de modèles simples, un certain nombre de ces sorties ne seront pas calculées par le modèle. L'adaptateur doit donc définir des stratégies, dans le cas ou d'autres composants auraient besoin de ces sorties. Selon les cas, ces stratégies peuvent être la considération de valeurs par défaut (souvent nulles), ou le calcul de certaines grandeurs par l'adaptateur lui-même, à partir d'autres sorties du modèle. Par exemple, l'adaptateur peut intégrer le vecteur vitesse du bateau afin de déterminer sa position si le modèle ne la fournit pas.

Le bateau étant le système contrôlé, ses sorties doivent comprendre toutes les données nécessaires au contrôleur, en termes de position, de vitesse et d'attitude. Les sorties comprennent donc les coordonnées géographiques du centre de gravité du bateau, son vecteur vitesse « sur le fond », c'est-à-dire exprimé dans le référentiel terrestre, les trois angles d'Euler, qui permettent d'orienter le bateau autour de son centre de gravité, ainsi que les vitesses angulaires du bateau, exprimées dans le référentiel du bateau. Les sorties des modèles peuvent être plus nombreuses, avec des accélérations de plateforme, des vitesses verticales ou transversales. Comme ces informations ne sont pas prises en compte dans les lois de pilotage actuelles, ni dans les développements proches, elles ne figurent pas dans le standard actuel, mais pourront faire l'objet d'extensions futures.

Les entrées doivent comprendre tous les éléments externes au bateau susceptibles d'avoir une influence sur son comportement. On y distingue deux catégories : les éléments d'environnement

Entrées			Sorties		
Grandeur	**Unité**	**Nombre de données**	**Grandeur**	**Unité**	**Nombre de données**
Angle de barre	degré	1	Position	degré	2
Vitesse du vent	nœud	1	Angles d'Euler	degré	3
Direction du vent	degré	1	Vitesse sur le fond	nœud	1
			Route sur le fond	degré	1
			Vitesses angulaires	degré/sec	3
			Angle au vent	degré	2
			Vitesse du vent	nœud	1

TABLE 8.2 – Entrées et sorties standard des modèles de bateaux

et les actions du contrôleur. Dans le cas de l'environnement, comme expliqué ci-dessus, seul le vent est considéré, et le standard des entrées pour le bateau correspond évidemment au standard des sorties pour l'environnement, à savoir la vitesse et la direction du vent. Pour les actions du contrôleur, comme les pilotes actuels n'agissent que sur la barre, une seule donnée est considérée : l'angle de barre effectif.

Les données de vent font l'objet d'un traitement particulier. En effet, elles sont produites par un composant dédié et figurent à ce titre dans les entrées du bateau. Cependant, il est fréquent en voile d'exprimer le vent dans le référentiel du bateau, et non dans un référentiel terrestre. La plupart des pilotes automatiques proposent par exemple des lois de régulation autour de l'angle entre l'axe du bateau et le lit du vent (régulateur d'allure). Comme expliqué dans la partie 2.2.1, il est d'usage de distinguer le vent réel du vent apparent. Si la vitesse du vent réel est identique quel que soit le référentiel d'expression, l'angle au vent réel (TWA), l'angle au vent apparent (AWA) ainsi que la vitesse du vent apparent (AWS) dépendent de l'orientation et de la vitesse du bateau. Ces trois données sont donc considérées comme des sorties du bateau. Dans le cas où le modèle de bateau ne les fournit pas, elles sont calculées par l'adaptateur à l'aide de formules trigonométriques classiques, afin que les autres éléments du simulateur puissent y avoir accès.

Le tableau 8.2 synthétise l'ensemble des grandeurs qui constituent le standard des entrées / sorties des modèles de bateaux, ainsi que leurs unités d'expression. Ces données sont ainsi disponibles pour l'ensemble des autres composants du simulateur, quel que soit le modèle de bateau utilisé.

L'interface *boatAdaptation*

A l'image de l'interface *windAdapation*, l'interface *boatAdaptation* assure la communication entre un modèle de bateau et son adaptateur, à la fois lors de l'initialisation (négociation de la forme des échanges par envoi de métadonnées) et lors de la simulation (échange de données). Cependant, les données échangées ici peuvent être de nature plus complexe qu'un champ de vecteur bidimensionnel. Des types supplémentaires sont définis pour permettre l'échange de vecteurs tridimensionnels (angle d'Euler, vecteur rotation) ainsi que des coordonnées de points géographiques.

De manière similaire à l'adaptation du vent, des stratégies temporelles et de référentiel sont déclarées, et une unité est déclarée pour chaque grandeur. En supplément, un ensemble de variables booléennes permet au modèle de déclarer quelles sont les données qu'il utilise et qu'il produit. A partir de ces informations, l'adaptateur sélectionne les fonctions à utiliser pour la communication durant la simulation. Il dispose à cet effet de plusieurs fonctions permettant l'envoi de données, le déclenchement du traitement (dans le cas où le modèle est synchrone), et la récupération des résultats.

Par ailleurs, deux stratégies supplémentaires sont déclarées : une stratégie spatiale qui indique si le modèle produit les données dans le référentiel du bateau (référentiel relatif) ou dans le référentiel terrestre (absolu), et une stratégie d'orientation qui permet de savoir si l'orientation du bateau est déterminée par le cap de son axe principal (*heading*), ou par le cap de sa route fond (COG, *Course Over Ground*).

L'ensemble des métadonnées échangées lors de l'initialisation est regroupé dans une variable structurée, qui est transmise à l'adaptateur. La figure 8.5 montre un exemple de déclaration de cette variable, qui correspond à la signature d'un modèle de bateau en particulier.

```
boatAdaptation::InitValues metadata;
metadata.isSynchrone = true;
metadata.isUsingGyro = false;
metadata.isProducingRelativeWind = false;
metadata.isUsingHeel = true;
metadata.isUsingTrim = false;
metadata.timeOrFrequencyValue = 10;
metadata.spatialReferenceValue = boatAdaptation::relative;
metadata.spaceCoordinatesStrategyValue = boatAdaptation::carthesian;
metadata.orientationStrategyValue = boatAdaptation::useHdg;
metadata.lengthUnitValue = boatAdaptation::nm;
metadata.speedUnitValue = boatAdaptation::kts;
metadata.angleUnitValue = boatAdaptation::deg;
```

FIGURE 8.5 – Exemple de déclaration des métadonnées liées à un modèle de bateau, en langage C++

8.3 Discussion sur le style architectural

Le style architectural AMSA consiste à séparer les principaux éléments du simulateur (bateau, environnement, pilote automatique, capteurs et actionneurs) dans des composants composites, avec les composants modélisant le bateau et l'environnement qui respectent le pattern d'adaptation. Comme les entrées et sorties de ces composants sont standardisées, le flux de données présent dans le *rootComponant* respecte un standard de communication pour les données liées à l'environnement et au bateau. Le pattern d'adaptation n'est pas proposé pour le composant du pilote automatique. Il s'agit en effet du composant que l'on cherche à mettre au point, et la définition d'une interface standard serait trop restrictive.

La méthode d'adaptation présentée pour l'adaptation du vent manipule un vecteur ou un champ de vecteur à deux dimensions. Elle adresse des problématiques d'unités et de repères d'expression pour des séries temporelles géolocalisées. Il est envisageable de réutiliser cette démarche pour généraliser l'adaptation de champs de scalaires ou de vecteurs tridimensionnels. On note d'ailleurs plusieurs similitudes avec l'interface d'adaptation du bateau, qui manipule également des champs vectoriels. Il pourrait être intéressant de mettre en commun les éléments d'adaptation des champs vectoriels, en proposant par exemple une interface générique qui serait étendue aux divers cas d'utilisation (vent, position, vitesse et attitude du bateau). Cependant, cela nécessiterait un mécanisme d'héritage des interfaces, mécanisme qui n'est pas encore présent dans le framework AMSA.

Le respect du style architectural décrit ci-dessus n'est pas obligatoire dans le framework AMSA. Le modèle à composants, les générateurs de code et les simulateurs peuvent être utilisés dans d'autres contextes. Plutôt que de constituer un élément contraignant, le style architectural propose un ensemble de bonnes pratiques qui permettent de résoudre les problèmes communs que sont la vision du système à divers niveaux de granularité et la gestion de l'hétérogénéité des composants. Comme certaines situations peuvent amener à y déroger partiellement (modélisation ou non des capteurs, intégration d'éléments logiciels ou matériels externes dans la boucle de contrôle...), le framework n'impose aucune construction prédéfinie.

Chapitre 9

Les scénarios

Les chapitres précédents traitent des possibilités offertes par le framework AMSA pour créer des simulateurs de pilotage de voiliers. Pour exploiter ces simulateurs, il est nécessaire de pouvoir paramétrer des simulations, de les exécuter selon des scénarios souhaités, puis de collecter et d'interpréter les résultats. Ce chapitre décrit les possibilités offertes par le framework AMSA pour réaliser de telles opérations.

A partir d'un simulateur modélisé sous forme d'une configuration de composants, le framework propose des concepts qui permettent de créer et d'éditer un scénario de simulation, mais également d'enchaîner plusieurs simulations pour balayer certaines valeurs paramètres. A un scénario sont associés des observateurs, dont le rôle est de collecter les résultats des simulations sous une forme facilement exploitable. Ces différents éléments sont décrits dans le méta-modèle *AMSA-Scenario*.

9.1 Description des scénarios

L'objectif premier d'un scénario est de pouvoir paramétrer, lancer et arrêter des simulations selon un enchaînement établi. Le scénario constitue un point d'accès au simulateur et règle la simulation. Il permet de modifier les grandeurs caractéristiques de la simulation au cours de celle-ci : conditions initiales, consigne de régulation, paramétrage des composants et durée de la simulation.

9.1.1 La structure

Un *Scenario* (*cf.* figure 9.1) décrit le déroulement d'une simulation. Il est rattaché à un simulateur donné, un lien existe donc entre le *Scenario* et le *rootComponent* (de type *CompositeComponent*) du simulateur. Un *Scenario* est composé de deux types d'éléments : des initialiseurs de paramètres (*ParamInitializer*) et des événements (*Event*).

Comme leur nom l'indique, les initialiseurs sont en charge d'initialiser les paramètres des composants avant la simulation. Ils permettent de mettre le simulateur dans la configuration souhaitée. Les événements interviennent eux durant la simulation. Chaque événement définit un temps auquel il sera considéré. L'unité temporelle est définie de manière homogène pour tout le scénario par l'attribut *timeUnit*, sa valeur par défaut étant la seconde.

Une fois configurée, l'exécution d'une simulation se déroule ainsi (on suppose ici que la configuration des composants qui constitue le simulateur a déjà été créée) :

1. les initialiseurs de paramètres sont exécutés,
2. les méthodes d'initialisation (*initialize()*) des composants sont exécutées,
3. l'horloge du simulateur est activée, la simulation démarre au temps t = 0,
4. à chaque pas de temps, les méthodes *doStep()* des composants sont exécutées,
5. dès que l'horloge atteint le temps renseigné par un événement, celui-ci est interprété dans l'instant,
6. un événement spécial (*StopEvent*) arrête l'horloge et met fin à la simulation,
7. les méthodes de finalisation (*finalize()*) des composants sont exécutées.

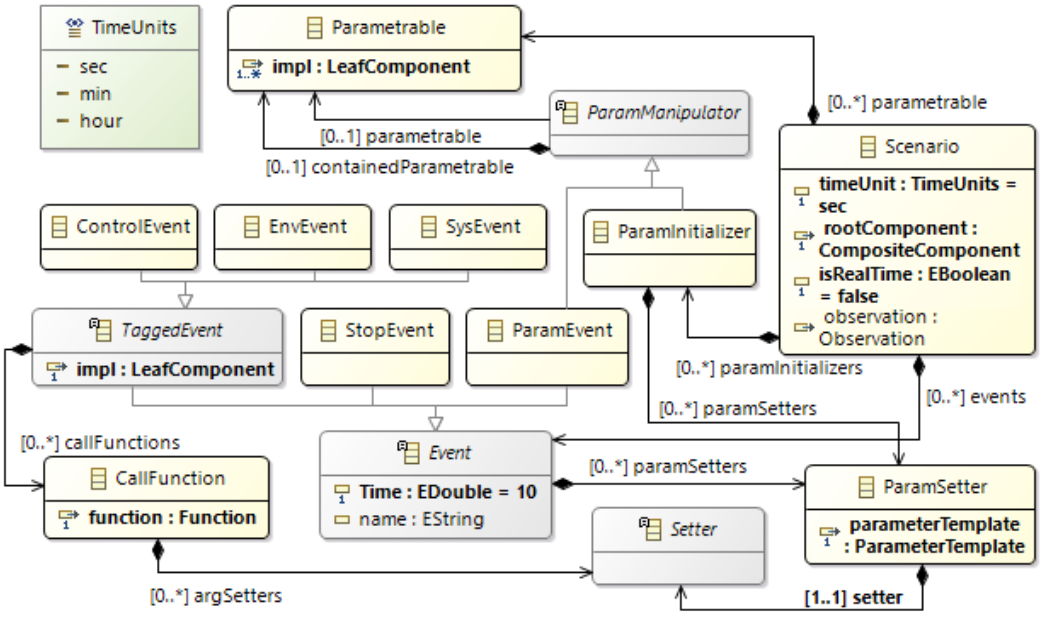

FIGURE 9.1 – Extrait du méta-modèle *AMSA-Scenario* : la structure

Comme le comportement des méthodes d'initialisation peut dépendre de la valeur des paramètres (par exemple pour l'ouverture d'un fichier dont le nom est paramétré), il est important que les initialiseurs du scénario interviennent avant l'appel aux méthodes d'initialisation. Cependant, ces méthodes sont également en charge d'initialiser les paramètres avec les valeurs renseignées lors de l'instanciation des composants. Un mécanisme de protection est donc nécessaire pour éviter que les paramètres ne soit initialisés deux fois, et c'est la valeur renseignée dans le scénario qui prévaut.

9.1.2 Les initialiseurs de paramètres

Les initialiseurs agissent sur des groupes paramétrables (*ParametrableGroup*), qui sont constitués d'une ou plusieurs références vers des composants du simulateur. Une contrainte *OCL* impose que tous les composants d'un même groupe soient issus du même template. Ils possèdent ainsi les mêmes paramètres. A travers le *ParametrableGroup*, il est possible de les manipuler de manière conjointe. On peut ainsi agir sur une, plusieurs ou toutes les instances d'un template. Selon la convenance de l'utilisateur, un groupe peut être déclaré en début de scénario, ou lors de la définition du *ParamInitializer*.

Un initialiseur contient un ou plusieurs *setters* de paramètres (*ParamSetter*). Un *ParamSetter* est associé à un paramètre (ou plus exactement au template d'un paramètre), et à une nouvelle valeur pour ce paramètre. Le changement concerne toutes les instances de composants rattachées au groupe paramétrable concerné. La valeur est fournie par un objet de type *Setter*, dont le comportement est détaillé au paragraphe 9.2.1.

9.1.3 Les événements

Les événements interviennent durant la simulation. On distingue trois principaux types d'événements, en fonction du type d'action qu'ils ont sur le simulateur :

Les *ParamEvent* qui décrivent des changements de valeurs de paramètres ;

Les *TaggedEvents* qui décrivent des appels de fonction, parmi les fonctions présentes dans les interfaces de marquage ;

Les *StopEvent* qui mettent fin à la simulation.

Les *ParamEvents* agissent de manière identique aux initialiseurs, à ceci près que leur action survient à un instant identifié au cours de la simulation : ils s'adressent à un groupe paramétrable par le biais de *ParamSetters*, et permettent ainsi de modifier la valeur des paramètres des composants

rattachés à ce groupe. Ce sont donc des événements pouvant concerner l'ensemble des composants disposant de paramètres.

Les *TaggedEvent* utilisent les interfaces de marquage pour identifier les composants auxquels ils s'adressent. Ainsi, on retrouve trois types de *TaggedEvents*, qui correspondent aux trois types d'interface de marquage (qui ont été présentés au paragraphe 8.1.1) : le type *SysEvent* permet d'adresser les composants qui modélisent le système contrôlé, le type *ControlEvent* cible les composants constitutifs du pilote, et enfin le type *EnvEvent* concerne les composants d'environnement.

Un *TaggedEvent* modifie toujours un et un seul composant. Il est possible de déclarer plusieurs événements au même instant si l'on souhaite modifier plusieurs composants de manière simultanée. Lors de l'exécution d'un *TaggedEvent*, il est possible de modifier les valeurs des paramètres du composant (en ayant recours à un *ParamSetter*), mais également d'appeler les méthodes de l'interface de marquage qui sont implémentées par le composant, via un objet d'appel (*CallFunction*). On peut ainsi déclencher des actions spécifiques au type d'événement (par exemple une modification de l'environnement pour les *EnvEvents*, un changement de loi de régulation ou de consigne pour les *ControlEvents*). Selon le prototype des méthodes, il peut être nécessaire de transmettre un ou plusieurs arguments lors de l'appel. La valeur de ces arguments est contenue dans un objet de type *Setter*.

9.2 Les balayages

Lors de la simulation, il est fréquent que l'utilisateur souhaite tester plusieurs valeurs pour un paramètre, dans le but d'évaluer son influence sur le contrôle. Bien qu'il soit possible d'éditer plusieurs fois le même scénario en changeant manuellement la valeur du paramètre à tester (et ainsi lancer autant de simulations que de cas à tester), le méta-modèle *AMSA-Scenario* propose des mécanismes de balayage de paramètres pour automatiser ce processus et éviter une fastidieuse exploration manuelle.

9.2.1 Les *setters*

Comme nous l'avons vu précédemment, les *setters* sont des objets qui permettent de modifier la valeur d'un paramètre de composant, ou la valeur d'un argument lors de l'appel d'une fonction. La figure 9.2 détaille la partie du méta-modèle *AMSA-Scenario* relative aux setters. On constate que ceux-ci peuvent être de deux types différents : simple (*SimpleSetter*) ou à balayage (*SweepedSetter*).

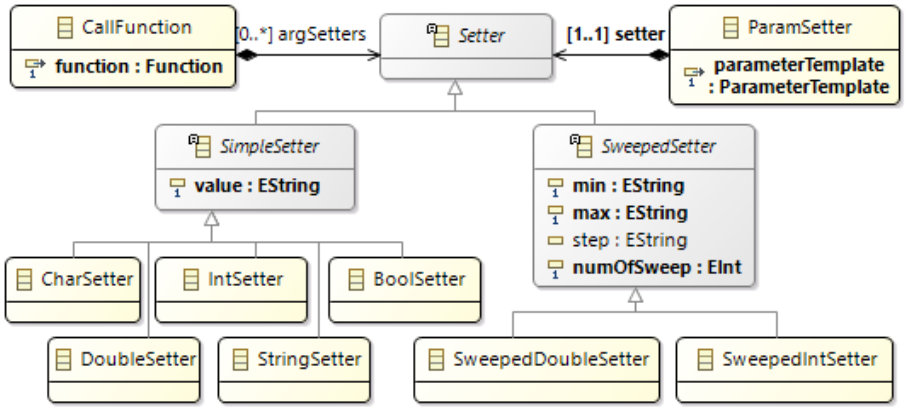

FIGURE 9.2 – Extrait du méta-modèle *AMSA-Scenario* : les *Setters*

Les *setters* simples sont caractérisés par une seule valeur. Leur rôle est d'assigner cette valeur au paramètre ou à l'argument de fonction auquel ils sont liés. Les *setters* à balayage, comme leur nom l'indique, proposent un balayage de valeur à assigner au paramètre ou à l'argument. Au lieu de définir une valeur unique au *setter*, on lui indique des bornes minimum et maximum, ainsi que le cardinal de l'ensemble balayé (nombre de valeurs dans l'ensemble : *numOfSweep*). A la place du cardinal, il est possible de renseigner l'incrément entre deux valeurs (*step*), le cardinal est alors automatiquement calculé.

La sélection d'un sous-ensemble de valeurs par cette méthode n'est possible qu'avec un type numérique. Ainsi les *setters* balayés ne sont disponibles que pour les paramètres de type entier ou réel. A l'inverse, les *setters* simples concernent tous les types primitifs.

9.2.2 Le déroulement d'un scénario avec balayage

Lorsqu'un scénario contient un ou plusieurs *setters* balayés, il est nécessaire de réaliser plusieurs simulations pour tester les différentes valeurs proposées par le setter. Une solution pourrait être de réaliser une seule simulation, qui enchaînerait les événements autant de fois que nécessaire pour tester toutes les valeurs balayées. Cependant, afin d'éviter les effets dus aux variations des conditions initiales (et donc l'influence des simulations précédentes sur la simulation actuelle), il est préférable de réinitialiser le simulateur à chaque nouvelle simulation. L'exécution d'un scénario se compose donc d'un enchaînement de simulations indépendantes, dont la structure a été décrite en 9.1.1. Cette approche permet d'effectuer des simulations en conditions strictement identiques (même temps, mêmes conditions initiales, même environnement), quelles que soient les valeurs des paramètres balayés.

Lorsqu'un scénario contient plusieurs *setters* balayés, toutes les combinaisons de valeurs possibles sont simulées. Pour pouvoir identifier les simulations et les valeurs des paramètres qui leurs sont associées, un fichier de rapport est créé lors d'une première phase d'exécution du scénario, appelée *pre-processing*. C'est également lors de cette phase que sont créés les composants d'exécution qui constituent le simulateur. Pour chaque *setter* balayé, on détermine le nombre de points de balayage, ainsi que la valeur du paramètre pour chaque point.

Une fois que l'ensemble des simulations a été effectué, le fichier de rapport est complété avec des résultats calculés par les observateurs, durant une phase de *post-processing*. Comme indiqué sur la figure 9.3, l'exécution complète d'un scénario se résume donc à trois étapes : *pre-processing*, enchaînement des simulations, puis *post-processing*.

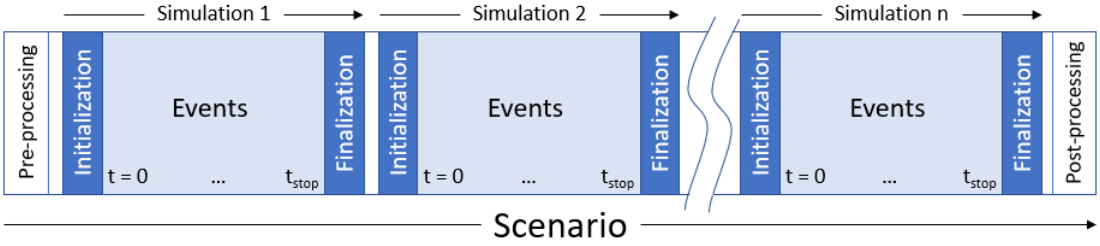

FIGURE 9.3 – Enchaînement de l'exécution d'un scénario

9.3 Sorties et critères d'évaluation

Au travers des mécanismes de paramétrage et de balayages, les scénarios permettent de tester autant de situations que souhaité. Afin de pouvoir exploiter les résultats de ces simulations, il est nécessaire de disposer d'un mécanisme de visualisation des résultats et de comparaison entre plusieurs simulations. Le méta-modèle *AMSA-Scenario* contient des éléments, visibles sur la figure 9.4, qui permettent de créer des observateurs liés à la simulation, dont le rôle est de collecter et de mettre en forme les résultats souhaités.

Les résultats sont de deux types : les séries temporelles, qui sont naturellement utilisées pour garder l'historique du flux de données durant la simulation, et les critères, qui sont des grandeurs scalaires résultant d'opérations mathématiques à partir des séries temporelles.

9.3.1 Les séries temporelles

Toute donnée qui transite dans le *dataflow* du simulateur peut être observée. La variation de la donnée au cours du temps crée une série temporelle qui permet d'obtenir la valeur de la donnée à chaque pas de simulation. La taille de la série temporelle dépend du temps total de simulation, ainsi que de la fréquence de calcul de la donnée.

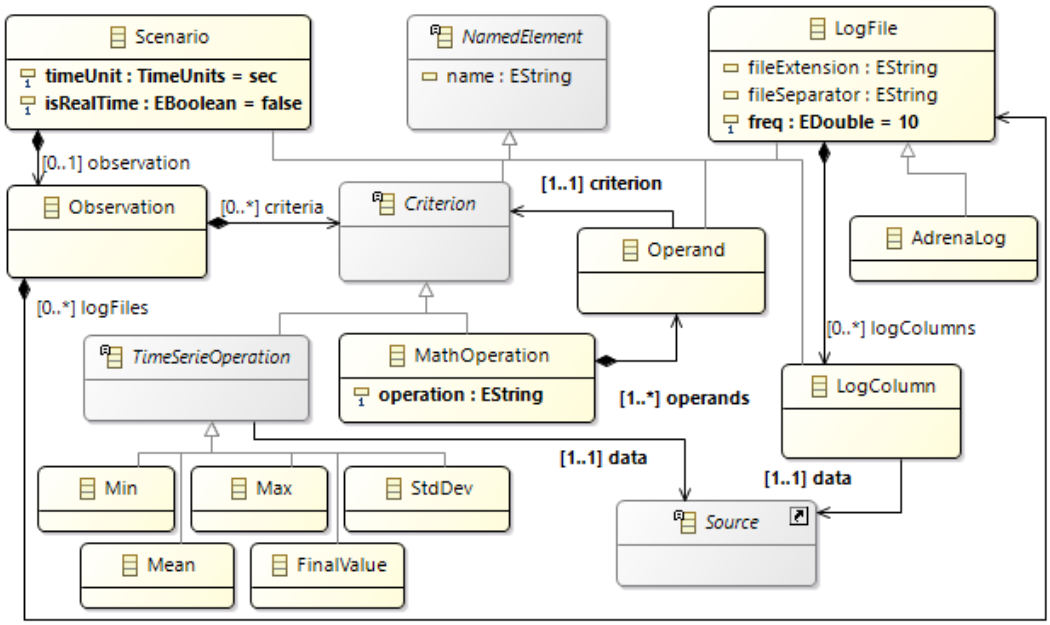

FIGURE 9.4 – Extrait du méta-modèle *AMSA-Scenario* : les observateurs

Les observateurs de séries temporelles (*LogFile*) permettent d'enregistrer une ou plusieurs séries (*LogColumn*) dans un fichier. Comme indiqué sur la figure 9.4, la série temporelle est directement liée à une donnée du simulateur (l'objet source du méta-modèle *AMSA-Component* regroupe l'ensemble des objets susceptibles de produire une donnée). L'extension du fichier ainsi que le séparateur utilisé entre les données peuvent être spécifiés. Il est ainsi possible de produire les résultats dans un format standard tel que le CSV. Un observateur spécialisé, *AdrenaLog*, permet quant à lui de stocker les séries temporelles au format *trz* utilisé par le logiciel de navigation *Adrena* [Adr18], afin de faciliter la visualisation des trajectoires simulées.

Dans le cas d'un scénario comprenant des balayages de paramètres, l'observateur de série temporelle produit un fichier par simulation, le fichier étant identifié par un nom suivi d'un numéro de simulation. La correspondance entre ce numéro de simulation et les valeurs de paramètres utilisées est faite dans le fichier rapport généré lors de l'étape de pre-processing. L'enregistrement des séries temporelles permet ainsi de rejouer une simulation particulière en affichant l'ensemble des données qui ont été enregistrées.

9.3.2 Les critères de performance

Si les séries temporelles sont nécessaires pour conserver l'historique des simulations, elles ne sont pas adaptées pour comparer rapidement les performances de diverses configurations. Pour cela, le méta-modèle propose un second type d'observateur sous forme de critères (*Criterion*). Il s'agit d'opérations mathématiques permettant de qualifier la simulation. On distingue les critères qui résultent directement d'une opération mathématique appliquée à une série temporelle (*TimeSerieOperation*) de ceux qui découlent d'une combinaison d'autres critères (*MathOperation*).

Dans le cas des critères relatifs aux séries temporelles, cinq opérations de base sont disponibles : le maximum, le minimum, la moyenne, l'écart-type, et la dernière valeur. Le choix du critère dépend de la nature de la série temporelle. Ce choix est laissé à l'utilisateur lors de l'édition du scénario, car il relève d'une expertise métier et de l'objectif de la simulation.

L'objet *MathOperation* a un comportement opaque car lié au domaine métier. Comme l'ensemble des opérations mathématiques envisageables n'est pas modélisé, il s'agit d'une opération en boîte noire (déclaration d'une chaîne de caractères qui sera insérée dans le code généré pour effectuer l'opération désirée). Il peut par exemple permettre de calculer des valeurs dérivées, en combinant les résultats d'autres critères.

9.4 Discussion sur les scénarios

En permettant de construire et d'éditer des simulations structurées, le mécanisme des scénarios facilite l'utilisation des autres éléments du framework AMSA. En effet, une fois un simulateur modélisé, le scénario constitue le principal point d'interaction entre l'utilisateur et le simulateur. Grâce au scénario, le simulateur peut être configuré pour réaliser des simulations en fonction des objectifs de l'utilisateur.

L'intégration du balayage de paramètres dans les scénarios permet en outre de générer rapidement des situations variées qui permettent d'explorer des valeurs de paramètres pour des situations données. Cela est particulièrement utile lors de la mise au point de lois de contrôle, mais également pour l'identification de modèles de bateaux ou d'environnement, lorsque l'on cherche à reproduire des situations et des comportements réels.

A l'instar des autres éléments du framework AMSA, un langage spécifique et son éditeur sont associés au méta-modèle *AMSA-Scenario*. Il s'agit d'un éditeur textuel, proposant une syntaxe adaptée à la création et à l'édition de scénarios de simulation. Ceci est détaillé dans le chapitre suivant, qui traite de l'outillage du framework.

Chapitre 10

L'outillage et l'utilisation du framework

Maintenant que les principes du framework AMSA ont été présentés, ce chapitre s'intéresse aux différents outils qui permettent de créer des simulateurs grâce au framework AMSA, depuis la création de modèles jusqu'à l'exécution du simulateur. L'ensemble de ces outils s'appuie sur l'environnement *Eclipse*, et en particulier sur les solutions développées dans le cadre de l'*Eclipse Modeling Project* [Ecl19b].

10.1 Les éditeurs de modèles

10.1.1 Interfaces et templates

Un éditeur est associé à chacun des quatre méta-modèles du framework AMSA. Pour l'édition des interfaces et des templates, une grammaire textuelle a été définie. Les éditeurs textuels de ces modèles se basent sur des langages spécifiques au domaine (*DSL*). Ces éditeurs sont basés sur le projet *Xtext* [Ecl19e]. Ils permettent d'instancier de manière efficace et concise les différents éléments des modèles considérés.

Pour les modèles d'interface, l'éditeur permet la déclaration de types énumérés (*enum*) ou structurés (*struct*), puis la déclaration des prototypes de fonction qui constituent l'interface. Ces prototypes peuvent utiliser les types primitifs ainsi que les types complexes déclarés dans la même interface (la réutilisation des types n'est possible qu'à travers une duplication du code). La figure 10.1 présente la déclaration de l'interface *windAdaptation*, qui assure la communication entre une source de vent et son adaptateur. Les types énumérés permettent de définir soit des stratégies (par exemple la stratégie temporelle *TimeStrategy* qui peut être synchrone, mixte ou asynchrone), soit des unités (par exemple l'unité angulaire *AngleUnit*, qui peut être le degré ou le radian). Les types structurés constituent des containers cohérents pour les données échangées. Les fonctions sont utilisées pour l'échange d'information, que ce soit à l'initialisation (*getInitValues()*, *getStartDate()*), ou lors de la simulation (*getWindData()*, *getPosition()*). On note que dans ce cas, certaines déclarations de structure ont été repliées afin d'augmenter la lisibilité de l'interface (lignes 15, 23 et 27). En répétant ce schéma, il est possible de déclarer plusieurs interfaces dans un même fichier.

```
Interface windAdaptation {
    Types {
        enum TimeStrategy {synchronous,mixed,asynchronous};
        enum TimeUnit {h,min,s,ms,hz};
        enum DateUnit {timeStamp,DDMMYYhhmmss,YYYYMMDDhhmmss};
        enum SpaceStrategy {localized,nonLocalized,spatialized};
        enum SpaceUnit {NM,LatMinSec,LatMinDixmin};
        enum ZUnit {m,ft};
        enum SpeedUnit {kts,mps,kmph};
        enum SpaceCoordinatesStrategy {carthesian,polar};
        enum AngleUnit {deg,rad};
        enum ZeroDirection {north,south,east,west};
        enum SensOfRotation {trigo,antiTrigo};
        enum DataOrigin {sensor,model,forecast,unknown};
        struct Date {..
        struct Position {
            double first_coord;
            double second_coord;
        };
        struct WindData {..
        struct InitValues {..
    }

    InitValues getInitValues();
    bool setStartDate(Date theDate);
    WindData getWindData();
    WindData getWindData(Date theDate);
    WindData getWindData(Date theDate, Position thePosition);
    Position getPosition();
}
```

FIGURE 10.1 – Un modèle d'interface représenté dans son éditeur

De manière similaire, l'éditeur de templates propose une syntaxe simple pour déclarer les différents constituants de ces derniers : comme le montre la figure 10.2, les mots clé *Provided* et *Required* permettent de renseigner quelles interfaces sont fournies ou requises par le template. La déclaration des entrées et sorties se fait à l'aide des mots *input* et *output*, tandis qu'un bloc *parameter* permet de déclarer le nom et le type des paramètres du template. On constate sur la figure 10.2 que le template *WindAdaptor* requiert l'interface *windAdaptation* précédemment déclarée, et qu'il possède deux entrées (*lat* et *lon*) ainsi que deux sorties *TWD* et *TWS*. Le template *PddPilot*, en plus de ses interfaces et entrées/sorties, déclare quatre paramètres, tous de type double, qui correspondent aux quatre coefficients de la loi de contrôle concernée. Comme l'éditeur construit de manière dynamique le modèle lors de l'édition, il signale via une icône les erreurs de syntaxe, mais également les incohérences avec les autres modèles, comme l'utilisation d'une interface qui n'a pas été déclarée au préalable.

```
Template PddPilot{
    Provided basicPilot
    input Hdg, dHdg, dheel, TWA
    output rudderCmd
    parameter{
        double Kc;
        double Kcap;
        double Klacet;
        double Kroulis;
    }
}

Template WindAdaptor {
    Required windAdaptation
    input lat, lon
    output TWD , TWS
}
```

FIGURE 10.2 – Un modèle de template représenté dans son éditeur

10.1.2 L'édition d'une configuration

Une fois que les interfaces et les templates ont été déclarés, l'instanciation des composants se fait de manière graphique, à l'aide d'un éditeur basé sur le projet *Sirius* [Ecl19d]. A chaque composant composite est associé un diagramme, qui présente le contenu du composite. Tous les composants enfants du composite y sont représentés. Un tel diagramme est visible sur la figure 10.3. Il s'agit d'un exemple de composant composite encapsulant une loi de pilotage. On y trouve une instanciation du template *PddPilot*, ainsi que deux composants qui permettant de dériver temporellement les données de cap et de gîte (*hdg* et *heel*). Un composant composite est également en charge de filtrer l'angle au vent (TWA) en fonction de la vitesse du bateau.

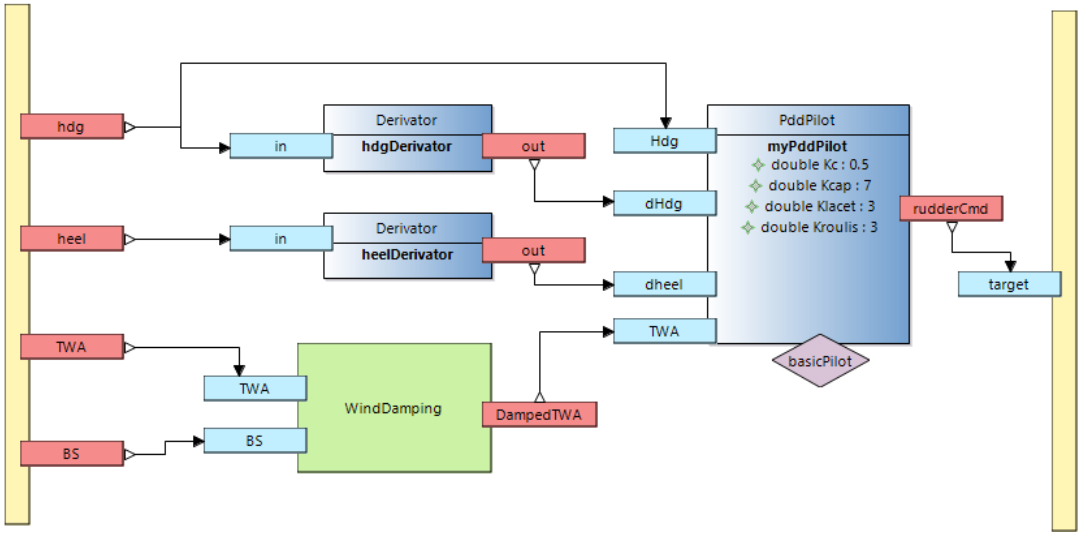

FIGURE 10.3 – Un diagramme de composant composite représenté dans son éditeur. On y voit les composants contenus (carrés verts pour les composites, bleu pour les atomiques) et leurs entrées / sorties (rectangles bleus / rouges sur les bordures). Le losange violet indique une interface de marquage, les bandes jaunes modélisent la frontière extérieure du composant

Les entrées et sorties sont visibles sous forme de ports attachés aux frontières des composants. Le principal avantage de cette visualisation graphique est que le flux de données est visible sous forme de flèches, qui relient les sorties aux entrées. La représentation graphique d'un composant composite respectant le pattern d'adaptation est visible sur la figure 8.3. Il s'agit de l'adaptation du vent, on y voit trois composants atomiques pouvant fournir des données de vent. Ils peuvent être reliés à l'adaptateur via l'interface de communication *windAdaptation*. L'édition du lien d'interface permet de sélectionner le composant que l'on veut utiliser.

Les bandes qui modélisent les frontières du composant composite portent les entrées et sorties correspondant aux ports composites liés à ce composant. Les ports composites d'entrée sont ici perçus comme des sorties (car ils fournissent une donnée venant de l'extérieur), tandis que les ports de sorties sont perçus comme des entrées (car ils requièrent une donnée pour la proposer à l'extérieur). Ces frontières permettent également la navigation entre les différents diagrammes qui constituent le modèle : un double-clic dessus ouvre le diagramme du composant parent, qui contient le composant actuel (on « remonte » d'un niveau). A l'inverse, un double-clic sur un composant composite ouvre son propre diagramme (on « descend » d'un niveau). L'éditeur permet ainsi de parcourir l'ensemble de la configuration en se déplaçant d'un diagramme à un autre de manière arborescente.

Pour manipuler les diagrammes, l'éditeur propose une palette de création graphique, qui permet d'instancier de nouveaux composants composites, de leur ajouter ou enlever des ports composites, d'instancier des composants atomiques à partir d'un template, et d'éditer les liens de données et d'interfaces entre les composants. L'instanciation de template est le seul moyen de créer des composants atomiques. Ceux-ci sont donc typés dès leur création, et il n'est pas possible de choisir ou de modifier le type d'un composant après son instanciation. Imposer un typage dès l'instanciation des composants représente une contrainte lors de l'édition de la configuration (obligation de déclarer un template avant de pouvoir instancier un composant), cette contrainte reste nécessaire pour permettre la génération de code sans avoir à vérifier la conformité des instances aux types déclarés.

10.1.3 Les scénarios

Les scénarios disposent d'un éditeur dédié, dont la grammaire textuelle a été optimisée pour alléger la déclaration des événements, tout en permettant de désigner de manière précise les composants à modifier. Un exemple est donné sur la figure 10.4. La première ligne permet de déclarer certaines propriétés du scénario : son nom, le *rootComponent* auquel il est rattaché, et son éventuel caractère temps réel (via la présence du mot clé *inRealTime*). La déclaration d'un scénario se fait ensuite en trois étapes :

l'initialisation des paramètres (lignes 2 à 11). Il est tout d'abord nécessaire d'indiquer le composant que l'on souhaite initialiser (comme par exemple le composant *myBoatAdaptor* à la ligne 7) avant d'indiquer la valeur initiale de certains paramètres (90 ° pour le paramètre *initialHdg* à la ligne 8).

l'enchaînement des événements (lignes 13 à 19). Pour chaque événement, l'instant d'exécution est indiqué après le mot-clé *at* et le composant concerné après le mot-clé *on*, comme on peut le voir à la ligne 16. Les *TaggedEvents* sont marqués par des mots-clé particuliers *onEnvironment*, *onController* et *onSystem*. Ainsi l'événement de la ligne 13 concerne un composant fournissant une interface de marquage de type *EnvironmentIntf*. Il contient un appel à une méthode de cette interface à la ligne 14, méthode demandant une rotation gauche du vent de 10 ° avec une période transitoire de 3 secondes.

la déclaration des observateurs (lignes 21 à 28). La ligne 22 correspond à la déclaration d'un critère maximum sur la SOG du bateau, tandis que les lignes 23 à 26 permettent de configurer un fichier de sortie avec une fréquence d'enregistrement, un nom et un séparateur de données (ligne 23). Ce fichier contient trois colonnes, dont le titre et la provenance des données sont indiquées aux lignes 24 à 16.

```
1  Scenario KcTuning for Demo inRealTime {
2      Initialize {
3          on Demo.Pilot.myPddPilot {
4              PddPilot.setPoint = 90;
5              PddPilot.Kc =  [0.05| 0.15 step 0.01];
6          }
7          on Demo.Boat.myBoatAdaptor{
8              BoatAdaptor.initialHdg = 90.0;
9              BoatAdaptor.initialSpeed = 10.0;
10         }
11     }
12
13     at 50 onEnvironment Demo.Environnement.WindPattern
14         WindPatternInterface.turnLeft(15, 3.0);
15
16     at 100 on Demo.Pilot.myPddPilot
17         PddPilot.setPoint = 120;
18
19     at 600 Stop
20 }
21 Outputs {
22     maxSpeed : Max of Demo.Boat.SOG;
23     TextLog at 10.0 Hz in "KcResults.csv" sep ';' {
24         add rudCmd : Demo.Pilot.target
25         add rudAng : Demo.Actuator.angle
26         add heel : Demo.Boat.heel
27     }
28 }
```

FIGURE 10.4 – Un scénario représenté dans son éditeur

On note qu'une syntaxe particulière, visible à la ligne 5, permet de définir des balayages directement lors de l'affectation d'une nouvelle valeur au paramètre. Ces balayages peuvent être déclarés dans la phase initialisation, ou lors d'un événement.

Il est important que le scénario désigne les composants qu'il souhaite modifier de manière non ambiguë. En effet, si le méta-modèle *AMSA-Component* interdit à deux composants d'avoir le même nom lorsqu'ils partagent le même parent, rien ne garantit l'unicité des noms au sein de la configuration globale. Afin de lever l'ambiguïté, l'éditeur utilise le nom complet de chaque composant, à savoir son nom précédé récursivement par les noms de ses parents, séparés par des

points. Sur le même principe, les noms de paramètres à modifier sont précédés par le nom du template qui les définit, tandis que les noms de fonctions sont précédés par le nom de l'interface à laquelle elles sont rattachées.

10.2 La création d'un simulateur

Trois grandes étapes sont nécessaires pour obtenir un simulateur de pilotage de voilier à partir du framework AMSA : l'édition des modèles, la génération et la complétion du code, la compilation et l'exécution de l'application. L'édition des modèles se fait à partir des éditeurs décrits dans la partie précédente. Nous détaillons maintenant les phases de génération, de complétion et de compilation du code.

10.2.1 Les étapes de génération

Tout comme l'édition des modèles, la génération de code se fait en quatre étapes, chaque étape étant liée à un type de modèle. Il est ainsi possible de mener une phase de génération dès qu'un modèle est édité, même si le modèle de la phase suivante n'est pas déterminé. Les générateurs se basent sur le projet *Acceleo* [Ecl19a], qui permet de générer du code dans un langage choisi par une transformation *model to text*. Le moteur d'*Acceleo* se base sur des templates de génération pour produire un ou plusieurs fichiers respectant le template et contenant les éléments du modèle. Le framework AMSA propose quatre points de lancement du générateur, un par génération. Il est ainsi possible de lancer la génération directement dans Eclipse, depuis les éditeurs de modèles.

La génération des interfaces

La génération des interfaces est une étape relativement simple, puisqu'elle consiste à traduire les types et prototypes de fonction qui ont été modélisés dans le langage cible, en l'occurrence C++.

Comme le concept d'interface n'existe pas en C++, des classes abstraites ne contenant que des méthodes virtuelles pures (sans implémentation) sont générées pour chaque interface. Un espace de nommage est également créé pour chaque interface. Il contient la classe abstraite, ainsi que l'ensemble des déclarations des types énumérés et structurés. Ce mécanisme d'encapsulation permet d'éviter les conflits de noms, en particulier si plusieurs interfaces définissent des types portant le même nom.

La génération des templates

Pour chaque template, une classe C++ est générée. Cette classe est abstraite, car elle contient des fonctions virtuelles pures (déclarées mais non implémentées) : une fonction de lecture pour chaque port d'entrée du template, et une fonction d'écriture pour chaque port de sortie. Ces fonctions sont virtuelles car les ports sur lesquels le comportement doit lire les données ne sont connus qu'à l'instanciation des composants. Ainsi, la génération des templates n'impose aucune politique de communication entre les composants. De même, pour chaque interface requise par le template, la classe contient un champ de type pointeur sur la classe d'interface correspondante.

Au niveau du comportement, les classes C++ générées contiennent le squelette des méthodes communes à tous les templates : un constructeur, un destructeur, ainsi que les fonctions *doStep, initialize* et *finalize*. Sont également présents les squelettes de l'ensemble des méthodes présentes dans les interfaces fournies par le template. Tous ces squelettes de méthodes ont une implémentation vide. Après la génération, l'utilisateur doit venir remplir le corps des fonctions afin de définir le comportement associé au template. Les entrées/sorties sont utilisables par appel aux méthodes abstraites, sans se soucier de leur implémentation. De même, l'utilisation des interfaces requises passe par les pointeurs sur ces interfaces, sans préoccupation de leur initialisation.

Le méta-modèle *AMSA-Template* ne permet de modéliser que la partie variable des templates de composants. Cependant, nous avons vu qu'un composant possède des propriétés et des méthodes implicites, qui ne sont pas modélisées. Au niveau du code C++, cette partie fixe est contenue dans une classe *LeafComponent* fournie par le framework, dont une représentation UML est donnée figure

10.5. Toutes les classes de template héritent de *LeafComponent*, et possèdent donc la propriété *freq* ainsi que les méthodes *doStep, initialize* et *finalize* qu'elles doivent implémenter.

La génération de template est la seule étape dans la création d'un simulateur où l'utilisateur doit intervenir dans le code généré, afin d'intégrer le comportement des composants qui instancieront les templates. Cela peut être fait soit en plaçant directement le code dans la classe générée, soit en faisant appel à des bibliothèques externes, cette seconde méthode étant à privilégier pour faciliter la réutilisation de code métier. Un mécanisme de protection fourni par le générateur *Acceleo* permet de sauvegarder le code inséré par l'utilisateur, afin d'éviter qu'il ne soit écrasé lors d'une nouvelle génération.

La génération des composants

Cette étape s'effectue à partir d'une configuration valide de composants, stockée dans un fichier *amsacore*. Avant la phase de génération, une analyse de l'ordre d'appel des composants est effectué, selon le mécanisme décrit en 7.3.3.

Une fois l'ordre d'appel déterminé dans chaque composite, un ordre est établi entre tous les composants atomiques en parcourant l'arbre de la configuration à partir du *rootComponent*. Pour chaque composant atomique, une classe C++ est générée. Cette classe hérite de la classe du template associé au composant. Elle contient une implémentation des méthodes abstraites d'entrée/sortie, ainsi qu'un mécanisme d'initialisation des pointeurs d'interfaces requises. En effet, c'est lors de cette étape de la génération que la configuration complète est connue. Pour chaque donnée, on peut désormais identifier le producteur et les consommateurs.

La politique de communication est définie au sein de ces classes générées : une variable tampon de type double est déclarée pour chaque donnée de sortie du composant. La fonction d'écriture de la donnée (publication par le composant d'une nouvelle valeur sur la sortie) vient écraser la valeur précédente, tandis que la fonction de lecture renvoie la valeur courante du tampon lorsqu'elle est appelée par les composants consommateurs de la donnée.

Lors de la génération, les composants composites disparaissent. L'ensemble des composants atomiques est remis au même niveau. La communication se fait directement, sans passer par les intermédiaires que représentaient les données et les ports composites. Ainsi, pour lire une entrée, un composant va directement appeler la méthode du composant proposant la sortie à laquelle l'entrée était reliée, souvent de manière indirecte, dans la configuration. Comme une classe C++ est créée pour chaque composant atomique, ces classes sont des singletons, elles ne sont instanciées qu'une seule fois. Si plusieurs composants ont été créés à partir du même template, des classes différentes sont créées, héritant de la même classe, la classe associée au template.

Le seul composant composite qui subsiste après la génération est le *rootComponent*. Il joue le rôle de *Factory*, et contient donc les références vers les instances des classes des composants atomiques, ainsi que des méthodes permettant de créer, d'initialiser et de détruire ces instances. Dans toutes ces méthodes, l'ordre d'appel précédemment déterminé est respecté.

La génération du scénario

La dernière étape permettant d'obtenir un simulateur exécutable est la génération du scénario, qui se déroule en trois temps :

- la génération des classes C++ correspondant aux observateurs déclarés dans le scénario (qui sont instanciées de manière identique aux composants du simulateur),
- la génération de la classe C++ contenant le scénario à proprement parler : elle contient un pointeur vers le *rootComponent* du simulateur, ainsi que des méthodes pour générer les matrices de paramètres balayés, pour initialiser le simulateur et lancer autant de simulations que nécessaire pour réaliser le scénario,
- un *launcher*, en charge d'instancier la classe de scénario. Il constitue le point d'entrée de l'application exécutable.

Les différents événements sont encapsulés dans une méthode de la classe *Scenario*. Cette classe est également en charge de l'instanciation de l'horloge qui cadence les simulations. A chaque tick, l'horloge interroge le scénario sur d'éventuels événements à exécuter. Une fois que ceux-ci ont été

traités, les composants sont réveillés dans l'ordre déterminé lors de la génération de la configuration, selon leur fréquence d'activation. La classe scénario assure l'enchaînement des trois étapes d'exécution : le *pre-processing*, l'enchaînement des simulations, puis le *post-processing*.

10.2.2 La gestion du temps

En plus du code généré, le framework propose un ensemble de classes C++ indépendantes des modèles, qui permet de mettre en œuvre les composants lors de la simulation. Un diagramme UML de ces classes est présenté figure 10.5. On y trouve notamment les classes de gestion du temps, héritant d'un même parent *Clock*. *RealTimeClock* permet de conduire des simulations en temps réel, tandis que *SimulatedTimeClock* est utilisé pour le temps simulé. On trouve également la classe *TimeableObject* proposant la méthode *dostep()*. Elle définit un type générique d'objets pouvant être manipulés par une horloge. Enfin, la classe *LeafComponent*, dont héritent tous les templates de composants générés, définit les méthodes d'initialisation et de finalisation, permettant ainsi de manipuler les composants de manière générique.

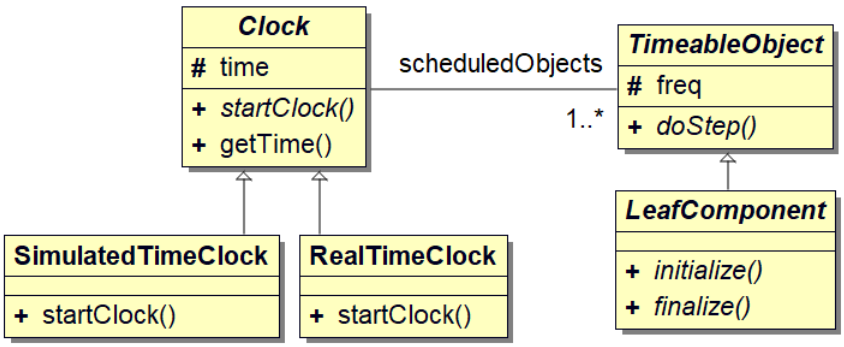

FIGURE 10.5 – Diagrame UML simplifié des classes C++ fournies par le framework AMSA

L'instanciation de l'horloge est à la charge du *launcher*, qui a lui-même été généré par le scénario. En fonction du caractère temps réel ou temps simulé de la simulation, indiqué par l'utilisateur lors de l'édition du scénario, le *launcher* choisit d'instancier soit la classe *RealTimeClock*, soit la classe *SimulatedTimeClock*.

L'implémentation des deux horloges respecte les principes présentés au paragraphe 7.3.4. L'algorithme de fonctionnement est identique dans les deux cas : lors du démarrage de l'horloge par la méthode *startClock()*, on relève la fréquence d'exécution de l'ensemble des *TimeableObjects* auxquels est reliée l'horloge. Ces fréquences sont converties en période (en milliseconde), et on en prend le PGCD pour déterminer l'espacement entre deux tics d'horloge. Lors d'un tic, les méthodes *doStep()* des objets sont appelées de manière séquentielle. La seule différence entre les horloges temps réel et temps simulé est l'espacement temporel entre deux tics d'horloge. Dans le cas du temps simulé, il n'y a pas d'espacement, les tics s'enchaînent en continu. Pour le temps réel, on fait l'hypothèse synchrone, qui présume que le temps d'exécution d'un pas de temps est inférieur à l'intervalle entre deux pas. On synchronise alors les ticks sur l'horloge du système d'exploitation qui héberge le simulateur, à l'aide des méthodes de la bibliothèque standard C++ regroupées dans l'espace de nommage *std* :: *chrono*.

10.2.3 La compilation et l'exécution du simulateur

Une fois les quatre phases de génération effectuées, on obtient un ensemble de classes et de fichiers C++, décomposable en quatre catégories :

- les classes entièrement générées, issues des phases de génération *interface, composant* et *scenario*,
- les classes de *templates*, générées puis complétées par l'utilisateur,
- les classes et méthodes fournies par le framework, détaillées au paragraphe précédent,
- les bibliothèques externes, ou *legacy code*, qui sont appelées depuis les classes de templates.

L'ensemble de ces éléments n'a pas d'autres dépendances que la bibliothèque standard C++ dans sa version C++11 (hormis bien sûr d'éventuelles dépendances dans le code ajouté par l'utilisateur), et peut donc être compilé pour un grand nombre de plateformes. La présence d'une méthode *main()* dans le *launcher* permet d'exécuter l'application obtenue. Les sorties du programme (écriture dans différents types de fichiers, affichage sur la console) dépendent des choix qui ont été faits lors de l'édition du scénario utilisé.

Il est important de noter que toute modification d'un des modèles (configuration des composants, scénarios...) nécessite une nouvelle étape de génération du code, et de compilation du simulateur. Ces étapes sont cependant entièrement automatisées, et aucune intervention dans le code n'est nécessaire, hormis pour modifier le comportement associé aux templates.

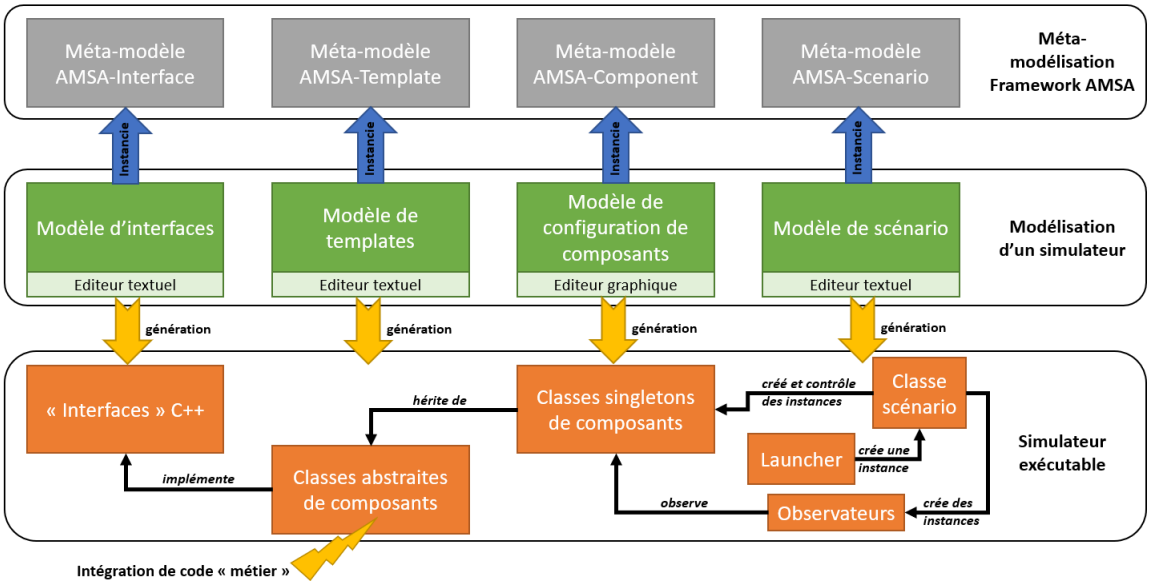

FIGURE 10.6 – Les étapes de modélisation et de génération d'un simulateur

La figure 10.6 récapitule les grandes étapes menant à la génération d'un simulateur exécutable. On y retrouve trois des niveaux de modélisation théorisés par l'OMG dans la spécification *Meta Object Facility* [OMG16], à savoir les méta-modèles, les modèles et la couche applicative. L'automatisation des différentes étapes de génération du framework AMSA permet au concepteur d'un simulateur de se concentrer sur les aspects fonctionnels des modèles qu'il utilise : grâce au pattern d'adaptation, des éléments hétérogènes peuvent être facilement assemblés, ce qui permet de construire des simulateurs spécialisés pour certains objectifs, comme l'évaluation ou la mise au point de lois de contrôle. Le développement ou la réutilisation de modèles existants pour le bateau, l'environnement et le pilote est toutefois nécessaire pour réaliser de telles simulations.

10.3 Cas d'utilisation du framework

Le développement d'un simulateur à l'aide des outils proposés par le framework est un processus décomposable en trois grandes étapes, qui correspondent à trois cas d'utilisation du framework, chacun en lien avec une expertise différente : la mise en place de l'architecture générale du simulateur, l'enrichissement de la bibliothèque de composants par des modèles spécialisés, et la conduite et l'exploitation des résultats des simulations pour répondre à un problème concret.

10.3.1 La mise en place de l'architecture

La première tâche est celle de l'architecte logiciel : il s'agit de définir une architecture du simulateur en agençant les divers composants de manière cohérente. L'utilisateur peut s'appuyer sur le style architectural présenté au chapitre 8 pour définir et standardiser les interfaces entre les composants.

Cette étape correspond à la production d'interfaces standard, des templates de composants qui

permettent de définir leurs interactions, et d'une configuration de composants qui décrit l'architecture du simulateur. Cette dernière est composée d'un premier niveau qui contient des composants composites et les données qu'ils échangent de manière standardisée. Chaque composite contient lui-même une structure arborescente plus ou moins complexe en fonction de la nature du phénomène modélisé.

L'architecture ainsi créée est générique vis-à-vis du domaine visé. Elle ne dépend pas d'un modèle de bateau particulier, ou d'une source de vent donnée, et doit pouvoir être utilisée pour intégrer des modèles de nature différente sans modification majeure, par le biais du pattern d'adaptation.

10.3.2 L'enrichissement de la bibliothèque des composants

Cette seconde étape consiste à créer une bibliothèque de composants spécifiques en encapsulant des modèles (existants ou développés pour l'occasion) dans des composants AMSA. Cette tâche est réalisée par les spécialistes des domaines considérés : mécaniciens pour les modèles physiques de bateau, météorologues ou statisticiens pour les modèles d'environnement et automaticiens pour les lois de contrôle.

Si l'architecture et les interfaces standards mises en place à la première étape sont pertinentes, l'encapsulation d'un modèle déjà existant dans un composant AMSA ne doit représenter qu'un faible effort d'intégration.

Cette étape permet la création d'un catalogue de composants basés sur des modèles spécialisés, permettant de répondre au besoin de simulations appliquées à des problèmes réels et concrets. Ces composants peuvent alors être intégrés dans l'architecture mise en place

10.3.3 La conduite et l'exploitation de simulations

Une fois les composants spécialisés intégrés dans l'architecture du simulateur, la dernière étape consiste à exploiter ce dernier en lançant des simulations et en en exploitant les résultats. Le mécanisme de scénario simplifie la mise en œuvre du simulateur en la rendant indépendante des étapes précédentes.

Les simulations peuvent être menées dans divers objectif : la mise au point de modèles de bateaux, le rejeu de situations réelles dans le but d'en saisir tous les aspects et de comprendre les réactions du pilote automatique, ou encore la mise au point de lois de contrôle. L'adaptation du simulateur au cas d'utilisation souhaité se fait soit en jouant sur l'architecture de celui-ci soit en modifiant le contenu des scénarios de simulation.

Cette étape de simulation est effectuée par les spécialistes du domaine pour lequel la simulation est menée. Ainsi les automaticiens peuvent utiliser les balayages pour mettre au point les coefficients de lois de contrôle, les architectes et ingénieurs navals peuvent étudier l'influence de certains paramètres dans leurs modèles physiques, tandis que la simulation en temps réel permet aux navigants d'appréhender directement par leur ressenti l'influence des différents réglages du pilote automatique.

10.3.4 Discussion sur les cas d'utilisation

On constate que l'enchaînement de ces cas d'utilisation ne suit pas toujours de manière linéaire les étapes de modélisation et de génération présentées dans ce chapitre. Il est fréquent de faire des aller-retours entre les différentes étapes, si l'on constate par exemple qu'il manque des entrées/sorties à un template lorsque l'on tente de l'instancier, ou encore qu'un paramètre que l'on souhaite changer dans un scénario n'a pas été modélisé. Un simulateur complexe se construit donc de manière itérative, en enrichissant progressivement les différents modèles.

Après un certain nombre d'itérations, l'architecture globale du simulateur varie peu, les modifications interviennent le plus souvent au sein des composants composites, et le respect du pattern d'adaptation permet de limiter les modifications architecturales (l'ajout d'une donnée ou d'un mode de communication non considéré est possible en enrichissant l'interface d'adaptation). De plus, les mécanismes de génération (et notamment de protection du code utilisateur lors de la génération) facilitent ces itérations, celles-ci se résument souvent à l'enrichissement du modèle via son éditeur, suivi d'une étape de génération.

Troisième partie

Évaluation

Chapitre 11

Création d'une bibliothèque de composants hétérogènes

L'application du style architectural proposé au chapitre 8 fournit une architecture de simulateur adaptée à la simulation de voiliers, ainsi que des interfaces standardisées pour l'adaptation de modèles de vents et de bateaux. A partir de ces interfaces, la mise en place de simulateurs requiert la constitution d'une bibliothèque de composants pour simuler le vent et les bateaux. Il est également nécessaire d'intégrer au simulateur les lois de contrôle que l'on cherche à mettre au point.

Ce chapitre présente la démarche adoptée pour constituer cette bibliothèque de composants. Il permet d'évaluer l'effort de développement nécessaire pour l'intégration de modèles d'environnement, de bateaux et de lois de pilotage au sein d'un simulateur.

11.1 Différentes sources de vent

Plusieurs approches sont possibles pour simuler le vent sur le plan d'eau sur lequel évolue le voilier. Certaines sont simples (hypothèse d'un vent constant), d'autres requièrent du matériel dédié (utilisation d'un enregistrement de vent par un capteur en situation réelle) ou encore font intervenir des données tierces (utilisation de prévisions météorologiques). Pour choisir le type de source de vent à utiliser, il est nécessaire d'identifier les besoins des simulations, de développer des modèles pour répondre à ces besoins, puis de les intégrer dans le framework sous forme de composants standardisés.

11.1.1 Expression du besoin

L'objectif principal du simulateur reste la mise au point de lois de contrôle. Deux principaux cas de figure peuvent alors se présenter : soit on cherche à rejouer une situation réelle, dans l'objectif de comprendre le comportement du contrôleur et de l'améliorer, soit on désire tester la loi de contrôle dans plusieurs contextes, que l'on choisit pour leur représentativité des événements réels.

Pour pourvoir rejouer une situation réelle, il est nécessaire de disposer d'un enregistrement du vent lors de cette situation, et de pouvoir lire cet enregistrement lors de la simulation. On note que le vent enregistré doit être indépendant des mouvements du bateau, il doit donc s'agir du vent réel.

Pour tester différents contextes de vent, il faut disposer d'un environnement de test paramétrable afin de pouvoir retrouver les comportements typiques du vent. Cela implique de pouvoir générer les variations temporelles du vent, qui respectent généralement des schémas bien identifiés, comme cela a été présenté au paragraphe 2.2.1. Ces schémas correspondent à des superpositions de phénomènes continus (variation lentes) et de phénomènes ponctuels (variation brutales). La simulation de tels schémas doit permettre la mise en place de tests systématiques sur les lois de pilotage. Cela permet également de répondre aux questions fréquentes que se pose le marin à ce sujet, comme par exemple : « Quel est le comportement du pilote dans un vent oscillant ? Comment réagit-il dans une rafale ou dans un refus ? ».

Deux besoins sont identifiés pour la mise au point de lois de contrôle : la possibilité de rejouer un enregistrement du vent, et la possibilité de simuler les différents phénomènes présentés sur le tableau 2.2.

11.1.2 Création des contextes

Trois approches sont envisagées pour créer des modèles de vent : une approche par série temporelle, une approche par paramétrage via le mécanisme de scénario, et une approche par la création d'une interface dédiée aux contextes de vents.

Les séries temporelles

La première approche consiste à créer une bibliothèque de vents sous forme de séries temporelles, à l'image d'un enregistrement. Chaque série temporelle contient un enchaînement de phénomènes ponctuels ou continus. La description des schémas de variations de vent est contenue dans la série temporelle. Cette approche revient à développer un composant par contexte de vent rencontré. La sélection de la série temporelle à utiliser se fait en éditant la configuration pour sélectionner le composant adéquat. Il est également possible de stocker les séries temporelles dans des fichiers, qui sont ensuite lus à l'image des enregistrements de vent réel. La sélection du fichier se fait alors à l'initialisation du scénario de simulation, la configuration n'est pas à modifier. Cette méthode présente l'avantage de faciliter la réutilisation d'un même contexte, par la réutilisation de la série temporelle. De plus, la sélection d'un contexte via le scénario de simulation est simple et immédiate.

Dans cette approche, les contextes obtenus ne sont pas du tout paramétrables, alors que les phénomènes à modéliser peuvent varier en intensité ou en durée. Il est ici impossible de modifier le vent à partir du scénario, à part en changeant de série temporelle. La durée de la simulation est également limitée par la durée de la série temporelle, même s'il est possible de définir des stratégies pour contourner cette limitation (comme par exemple le retour automatique en début de série).

Le paramétrage

Pour remédier au problème de paramétrage et offrir plus de souplesse dans la modification du vent via le scénario, il est possible de proposer un composant qui fournit un vent constant et paramétré. A chaque variation désirée on associe un événement dans le scénario, afin d'indiquer une nouvelle vitesse et une nouvelle direction de vent. Les schémas de variation du vent sont alors entièrement décrits dans le scénario de simulation.

Si cette méthode présente l'avantage de pouvoir paramétrer les variations, elle augmente notablement la complexité du scénario : les variations continues étant impossibles à paramétrer, il est nécessaire de faire des variations incrémentales fréquentes. De plus, les phénomènes périodiques telles les oscillations doivent êtres décrits « à la main » avec une duplication des événements à chaque période. La description d'un contexte de vent élaboré alourdit alors considérablement le scénario.

La définition d'une interface de vent

La troisième approche consiste à encapsuler les schémas de vent, qui relèvent de la connaissance métier, dans un composant dédié, et de créer une interface permettant l'accès à ces schémas depuis le scénario de simulation. Le rôle de cette interface n'est pas la communication entre composants du simulateur, mais bien la manipulation d'un composant d'environnement depuis le scénario. On a donc recours à une interface de marquage de type *EnvironmentIntf*.

L'interface *WindPatternInterface*, visible sur la figure 11.1, contient ainsi des fonctions permettant de définir un vent constant (*constantMode*) et de créer des oscillations sur la vitesse ou la direction (*setHarmonicSpeed* et *setHarmonicDirection*). Les arguments de ces fonctions permettent d'indiquer l'amplitude (en nœuds et en degrés) ainsi que la période (en secondes) de ces oscillations. De même, les phénomènes ponctuels peuvent être adressés par les méthodes de l'interface, que ce soit les rafales (*squareRise* et *rampRise* pour une augmentation progressive), les molles (*squareLull* et *rampLull*) ou encore les rotations (*turnLeft* et *turnRight*).

```
134 Environement WindPatternInterface {
135     void constantMode(double TWS, double TWD);
136     void setHarmonicSpeed(double ampl, double period);
137     void setHarmonicDirection(double ampl, double period);
138     void squareRise(double ampl, int duration);
139     void rampRise(int ampl, int duration, double rampDuration);
140     void squareLull(double ampl, int duration);
141     void rampLull(int ampl, int duration, double rampDuration);
142     void turnLeft(int angle, double speed);
143     void turnRight(int angle, double speed);
144     void ramp(double coef, int duration);
145 }
```

FIGURE 11.1 – L'interface *WindPatternInterface*

L'élaboration d'un contexte de vent se fait par appels aux méthodes de cette interface dans le scénario de simulation. Il est possible de changer le paramétrage du contexte via les arguments des fonctions, mais la gestion des événements eux-mêmes (variations, oscillations) est déléguée au composant.

11.1.3 Développement des composants AMSA

Chacune des trois approches exposées ci-dessus requiert un composants différents : le premier pour lire des séries temporelles, le deuxième pour fournir un vent constant paramétré, et le troisième pour générer du vent à partir de l'interface *WindPatternInterface*. Cette partie présente l'implémentation de ces trois composants.

En ce qui concerne les séries temporelles, le composant *FileWindReader* vient lire les informations de vent stockées dans un fichier à une fréquence connue, et les communique à l'adaptateur. Le format de stockage des séries temporelles est identique à celui utilisé par une centrale de navigation pour les enregistrements du vent. Ce composant peut donc également être utilisé pour rejouer une situation réelle à partir d'un log. Le template du composant ne déclare aucune entrée ou sortie, la communication se fait via l'interface *WindAdaptation*. Le template fournit également l'interface *basicWind*, une interface de marquage vide dont le but est uniquement d'identifier le composant comme source de vent. Le composant dispose d'un paramètre *fileName* de type *string* qui permet, lors de l'initialisation du composant, de choisir quel fichier lire. La complétion du code C++ généré est des plus simples : ajout de 6 lignes de code pour gérer un objet de type *fileStream* (ouverture du fichier dans la méthode *initialize()* et fermeture dans la méthode *finalize()*), et de 3 lignes pour venir lire les valeurs à chaque pas d'exécution.

Un composant de vent constant (*ConstantWind*) implémente l'interface *WindAdaptation*, et dispose de deux paramètres réels, respectivement pour la vitesse et la direction du vent. Ici encore l'interface de marquage *basicWind* est fournie par le composant. L'implémentation du comportement de ce composant est très simple, puisqu'il s'agit simplement d'envoyer la valeur des paramètres via l'interface de service (3 lignes de code ajoutées au fichier généré).

Dans l'objectif de pouvoir créer facilement des contextes de vent paramétrés, un composant *WindPattern* est créé. Il implémente l'interface *WindAdaptation*, mais également l'interface de marquage *WindPatternInterface*. Comme ce générateur et l'interface qu'il implémente ont été conçus spécialement pour faciliter la création de contextes de vent dans les scénarios, il n'existe pas de code métier qui décrit le comportement souhaité. Une implémentation minimaliste de l'interface est donc embarquée directement dans le composant, par l'ajout d'une cinquantaine de lignes de code.

Afin de disposer de trois sources potentielles de vent dans le simulateur, une instance de chacun de ces composants (*FileWindReader*, *ConstantWind*, *WindPattern*) est créée au sein du composant composite d'environnement. Ces instances, visibles sur la figure 8.3 peuvent être reliées à un adaptateur selon le pattern d'adaptation.

11.2 Différents modèles de bateaux

La précision d'un modèle de voilier, à savoir sa capacité à reproduire de manière fidèle le comportement d'un bateau réel, est un élément important qui conditionne l'utilisation possible du modèle. La construction d'un modèle analytique précis requiert une description complète des efforts aérodynamiques et hydrodynamiques qui s'appliquent sur le voilier, ainsi qu'une bonne connaissance des inerties et des amortissements du bateau. Cela implique un effort conséquent de développement du modèle, ainsi que d'importants moyens en termes de calcul pour la résolution des équations de la dynamique des fluides.

Cependant, il est possible d'obtenir des modèles de voilier nécessitant des efforts plus limités. Leur précision est moindre, mais peut être suffisante pour obtenir des résultats aidant à la mise au point de lois de contrôle. Dans cette partie, nous décrivons deux approches pour obtenir de tels modèles : l'utilisation d'un modèle analytique simplifié, et l'obtention d'un modèle paramétrique identifié à partir d'enregistrements sur un bateau réel.

11.2.1 Modèle analytique

La physique du modèle

Un modèle analytique de bateau a été proposé par L. Jaulin [Jau04]. Celui-ci est basé sur deux hypothèses simplificatrices. La première porte sur la réduction du nombre de degrés de liberté (DDL) : un mobile évoluant dans l'espace possède 6 DDL (trois translations et trois rotations), mais les DDLs qui ne jouent pas de rôle majeur dans le pilotage du bateau sont ignorés. Les seuls DDLs conservés sont les deux translations dans le plan horizontal et la rotation autour de l'axe vertical. On note que la rotation autour de l'axe longitudinal (qui correspond à l'angle de gîte) est négligée, ce qui constitue une simplification assez forte. La seconde hypothèse porte sur l'estimation des efforts : les forces appliquées sur les surfaces porteuses (voile et safran) sont supposées proportionnelles à la vitesse de l'écoulement et au sinus de son angle d'incidence, tandis que la traînée est supposée visqueuse (proportionnelle à la vitesse).

En appliquant les équations fondamentales de la dynamique sur le système ainsi simplifié, on obtient un système d'équations différentielles qu'il est possible d'intégrer numériquement pour obtenir le comportement du bateau. Le modèle proposé par L. Jaulin [Jau04] admet deux commandes : l'angle du safran (également appelé angle de barre) et l'angle d'ouverture de la voile. Pour se placer dans les conditions de simulation d'une loi de pilotage où seul l'angle de barre est commandé, on ajoute au système une équation supplémentaire qui stipule que l'angle d'ouverture de la voile est égal à la moitié du TWA.

On obtient ainsi un modèle analytique comportant 11 paramètres de masses, inerties, traînées et portances. Ses trois entrées sont la vitesse et la direction du vent réel ainsi que l'angle de barre. Ses sorties sont la vitesse du bateau et son cap.

L'intégration du modèle au framework

Un composant *AnalyticBoat* est créé pour encapsuler le modèle de bateau ainsi obtenu. Ce composant fournit l'interface *boatAdaptation* dans le but de pouvoir communiquer avec l'adaptateur universel. Il fournit également une interface de marquage vide, *BasicBoat*, afin d'être identifié comme un modèle de bateau. Les 11 paramètres du modèle sont déclarés dans le template du composant, ce qui les rend accessibles depuis les scénarios.

L'intégration du système d'équation permettant de calculer la vitesse et le cap du bateau à chaque pas de temps nécessite 12 lignes de code C++. Celles-ci sont directement insérées dans le corps de la méthode *doStep* de la classe générée à partir du template. Il est également nécessaire d'enrichir la classe avec des attributs contenant le cap, les vitesses et les accélérations linéaires et angulaires du pas de temps précédent. Cela passe par leur déclaration (5 lignes) et leur initialisation dans la méthode *initialize()* (5 lignes).

Comme le template implémente l'interface *boatAdaptation*, il est nécessaire de compléter l'implémentation des méthodes de l'interface, en particulier celle permettant la transaction du protocole de communication (13 lignes pour préciser les unités et conventions utilisées par le modèle) et celle permettant l'échange des données avec l'adaptateur lors de l'exécution (11 lignes). L'implantation

du modèle analytique dans un template AMSA requiert donc l'ajout de 46 lignes de code C++ au fichier généré. Parmi ces 46 lignes, 18 correspondent à des déclarations de variables. L'effort d'intégration à fournir est donc très modéré.

Détermination des valeurs de coefficients

Pour utiliser le modèle analytique afin obtenir des résultats représentatifs du comportement d'un bateau réel, il est nécessaire de disposer de valeurs de paramètres cohérents pour les masses, les inerties, les coefficients de portance et de traînée. Si certaines caractéristiques d'un voilier sont simples à estimer (comme sa masse ou ses dimensions), certaines sont moins triviales (coefficient de portance, traînée et amortissement). Il est possible d'identifier ces valeurs à partir du comportement réel d'un bateau, mais cela nécessite une série spécifique d'essais en mer [Bin+08].

L'approche utilisée pour trouver des valeurs de coefficients est plus empirique, elle repose sur l'appréciation de l'expérimentateur pour déterminer si la trajectoire du bateau est « crédible » dans un contexte de vent donné. Les paramètres de masse et de dimensions ont été renseignés à partir de valeurs courantes pour un monocoque de 60 pieds. Puis les possibilités de balayage offertes par les scénarios du framework AMSA ont été utilisées afin de déterminer des valeurs crédibles pour les paramètres inconnus, à savoir des valeurs permettant de retrouver un comportement de bateau qui ressemble à ce qu'on attend pour un monocoque de 60 pieds.

Cette méthode permet la mise au point rapide d'un modèle utilisable dans le simulateur, et permet de produire des résultats cohérents. Elle n'est cependant pas rigoureuse, car elle est basée uniquement sur l'appréciation de l'expérimentateur. Une manière de résoudre ce problème est d'utiliser les logs de navigation d'un bateau aux caractéristiques proches pour évaluer la capacité du modèle à reproduire des situations réelles. C'est une généralisation de cette approche qui est mise en œuvre dans l'élaboration de modèles paramétriques.

11.2.2 Modèles paramétriques

Les modèles analytiques sont simplifiés par des hypothèses fortes, qui conditionnent la forme des équations du modèle. De plus, ils sont souvent élaborés sur la base de petits monocoques, et ne prennent pas en compte les modifications de comportement apportées par l'arrivée des foils et des plans porteurs. Ainsi, les équations fournies par ces modèles ne sont pas forcément adaptées pour décrire le comportement d'un multicoque à foils tel le trimaran MACIF. Cette partie s'intéresse aux possibilités offertes par les modèles paramétriques pour le développement d'un modèle permettant de reproduire le comportement du trimaran MACIF.

Principe de l'approche

L'approche des modèles paramétriques consiste à ne pas faire d'hypothèses *a priori* sur la forme des équations qui régissent la dynamique du système et sur la valeur des coefficients de ces équations. Au contraire, des équations génériques sont utilisées, dont les coefficients (les paramètres du modèle) sont identifiés à partir de données enregistrées en conditions de navigation réelles.

Un modèle paramétrique est un modèle mathématique qui décrit le comportement d'un système en utilisant un nombre fini de paramètres. Pour les systèmes discrets à entrées multiples et sorties multiples (systèmes MIMO - *Multiple Inputs Multiple Outputs*), deux types de modèles paramétriques sont envisageables : les modèles d'état (basés sur un système d'état) et les modèles autorégressifs (basés sur un ensemble de fonctions de transfert). Comme la forme des équations n'est *a priori* pas connue, on utilise ici des modèles linéaires, qui sont plus simples à mettre en place et nécessitent des calculs peu coûteux. Une étude préalable à cette thèse a montré que les modèles d'état linéaires représentent de bons candidats pour la simulation du comportement de trimarans de course [Lav+16], c'est donc le type de modèle qui est utilisé ici.

Un modèle d'état linéaire est constitué de quatre matrices A, B, C et D dont les coefficients constituent les paramètres du modèle paramétrique. Un vecteur d'état représente l'état du système à un instant donné. L'ordre du modèle correspond à la taille du vecteur d'état. On peut résumer le système d'état linéaire discret sous forme de deux équations :

$$\begin{cases} X_{k+1} &= A \cdot X_k + B \cdot U_k \\ Y_k &= C \cdot X_k + D \cdot U_k \end{cases}$$

La première équation permet de déterminer le vecteur d'état X à l'instant $k+1$ à partir de l'état à l'instant k et des entrées U à l'instant k, tandis que la deuxième permet de calculer les sorties Y à partir de l'état et des entrées. Les coefficients des différentes matrices n'ont pas de signification physique, telle qu'une masse ou un amortissement. Ils sont identifiés à partir d'un log des entrées et sorties d'une navigation réelle : un mécanisme d'optimisation permet de trouver, pour un ordre de modèle donné, les valeurs de coefficients permettant de simuler les sorties les plus proches (au sens d'une norme) des sorties enregistrées.

A chaque navigation du trimaran, on dispose d'enregistrements des différentes données mesurées et calculées par la centrale de navigation (que l'on nomme généralement les *logs*). Ces données se présentent sous la forme de séries temporelles. Si l'on considère le voilier comme un système dynamique, certaines données sont perçues comme des entrées du système (le vent, l'angle de barre, les réglages du bateau), tandis que d'autres sont des sorties (la position et la vitesse, l'attitude). Une partie des données du log ne concernent pas directement la dynamique du bateau et sont ignorées (température de l'eau, pression atmosphérique, déformation de la plateforme...)

Pour simuler le comportement du trimaran MACIF, on considère trois entrées et trois sorties au modèle. Les entrées sont la vitesse et la détection du vent réel, ainsi que l'angle de barre qui correspond à l'action du contrôleur. Les sorties sont la vitesse, le cap et l'angle de gîte du bateau. Le fait de travailler avec le vent réel et non avec le vent apparent permet d'obtenir une meilleure précision des modèles [Lav+16].

Choisir un modèle linéaire pour modéliser le comportement d'un voilier peut paraître surprenant, car les équations qui régissent les écoulements fluides autour de celui-ci ne sont pas linéaires. Il est courant d'approximer la portance d'une voile comme étant proportionnelle au carré de la vitesse de l'écoulement. Pour remédier à cette source de non-linéarité dans le modèle, on ajoute en entrée du modèle une nouvelle série temporelle correspondant à la vitesse du vent réel élevée au carré.

Le domaine de validité du modèle

L'expertise métier des navigateurs nous révèle que de nombreux paramètres entrent en compte dans le comportement d'un trimaran de course : l'allure, la force du vent, les réglages des voiles et des appendices, l'état de mer. La combinaison de ces paramètres, dont certains sont discrets et d'autres continus, forme de nombreux contextes de navigation dans lesquelles la dynamique du bateau peut varier.

Un modèle paramétrique est toujours dépendant des données qui ont servi à l'identification de ses paramètres. En effet, par le processus d'identification, le modèle « apprend » les comportements dynamiques du système, mais seuls les comportements figurant dans le log d'identification peuvent être capturés par le modèle. Comme un log est toujours enregistré dans un contexte donné, il apparaît difficile de pouvoir construire un modèle décrivant l'ensemble du comportement d'un tel engin. Il est ainsi nécessaire de construire plusieurs modèles dans plusieurs contextes différents, on parle également de points de fonctionnement. Chaque modèle dispose d'un domaine de validité autour du point de fonctionnement. Ce domaine correspond à la région de l'espace des contextes (en particulier le vent) dans laquelle les résultats produits par le modèle restent cohérents.

Pour produire des résultats valables, une simulation utilisant un modèle paramétrique doit rester dans le domaine de validité du modèle. Cela permet également de justifier l'utilisation de modèles linéaires : lors des petites variations des éléments de contextes, l'approximation linéaire donne des résultats valides, mais ceux-ci peuvent devenir extrêmement faux lorsque l'on sort du domaine de validité du modèle. Pour pouvoir explorer différents contextes, on construit donc plusieurs modèles autour de différents points de fonctionnement, sans avoir la prétention d'explorer l'ensemble des possibilités.

Il peut être tentant de regrouper plusieurs modèles en un seul, afin de profiter d'un domaine de validité étendu. Cela nécessiterait de mettre au point des stratégies de transition entre les modèles, ce qui peut s'avérer complexe. De plus, les résultats produits dans les périodes de transitions n'auraient pas une grande pertinence. Il paraît donc plus judicieux d'exploiter le mécanisme des scénarios pour faciliter l'enchaînement de plusieurs simulations dans plusieurs contextes plutôt que d'enchaîner ces différents contextes dans la même simulation tout en sachant que les zones de transitions ne seront pas pertinentes.

La construction d'un nouveau modèle

La construction d'un modèle paramétrique nécessite avant tout de disposer d'un log de navigation dans le contexte dans lequel on souhaite réaliser le modèle. Il est important d'extraire une phase suffisamment longue pour permettre l'identification, mais sans changement brutal de contexte (rotation importante du vent, changement d'allure, virement de bord...).

Une fois le log choisi, des méthodes d'optimisation permettent de déterminer l'ordre du modèle et la valeur de ses paramètres. Ces méthodes sont basées sur la minimisation de la norme entre les sorties produites par le modèle et les sorties enregistrées dans les logs. Plusieurs critères sont ensuite disponibles pour évaluer la précision du modèle fourni [Lav+16].

Pour les besoins de cette étude, trois modèles du trimaran MACIF ont été identifiés autour de trois contextes différents. Le premier correspond à du près océanique (plus ouvert que le près serré) dans un vent moyen de 25kts. Le second et le troisième correspondent respectivement au *reaching* et au portant dans 20kts de vent. Dans tous les cas, l'état de mer correspond à la mer du vent, le bateau est mené en solitaire et réglé en conditions de course.

L'intégration au sein du framework

Un composant *ParametricBoat* est créé pour encapsuler les modèles paramétriques. Comme dans le cas du modèle analytique, ce composant implémente les interfaces *boatAdaptation* et *basicBoat*, qui lui permettent respectivement de communiquer avec l'adaptateur et d'être identifié en tant que modèle de bateau.

Les matrices des modèles paramétriques sont stockées dans un fichier ASCII, dans un format standardisé. Ce fichier peut également contenir des commentaires, qui permettent de préciser le contexte dans lequel le modèle a été identifié. Ainsi, le composant *ParametricBoat* ne contient qu'un seul paramètre de type *string*, qui décrit le nom du fichier de modèle à utiliser. Ce fichier est analysé lors de l'initialisation du composant, et les différentes matrices sont chargées en mémoire.

La manipulation des vecteurs et des matrices utilisés pour décrire le modèle d'état est déléguée à une classe C++ *StateModel*. La classe générée à partir du template est complétée avec les appels à cette classe *StateModel*, ainsi qu'avec l'implémentation des méthodes de l'interface *BoatAdaptation*, soit une trentaine de lignes de code ajoutées.

11.3 Différentes lois de pilotage

Une fois que l'on dispose de composants pour les modèles de vent et de bateau, il est nécessaire d'intégrer dans le simulateur la loi de contrôle que l'on souhaite mettre au point. Cette intégration dépend de la nature de la loi de contrôle :

- soit il s'agit d'une loi décrite par un algorithme ou une équation, que l'on peut alors intégrer dans un composant logiciel. C'est généralement le cas lorsque l'on développe une nouvelle loi ou que l'on cherche à paramétrer une loi connue ;
- soit il s'agit d'une loi embarquée dans un pilote existant, et dont on dispose d'une version logicielle ou matérielle. La loi est alors inconnue, la simulation a pour objectif de trouver les valeurs de paramètres optimales pour une situation donnée. Ce cas de figure apporte des problématiques supplémentaires : communication avec le logiciel (*software in the loop*) ou avec le calculateur (*hardware in the loop*) existant et gestion du temps réel dans le simulateur.

Cette section permet donc d'évaluer les capacités du framework AMSA en termes d'adaptabilité pour ces deux problématiques.

11.3.1 Lois de contrôle et actionneurs

Une loi de contrôle, quelles que soient sa nature et sa provenance, fournit une consigne d'angle de barre. Cette consigne est ensuite transformée en action (angle de barre effectif) par un actionneur, qu'il est nécessaire de modéliser. Il existe plusieurs possibilités pour modéliser l'actionneur, certaines très simples (hypothèse de l'actionneur idéal, l'angle de barre est égal à la consigne) et

d'autres plus complexes prenant en compte la dynamique de l'actionneur et les efforts dans la barre induits par l'action de l'eau sur le ou les safrans.

Comme les modèles de bateaux dont nous disposons ne permettent pas d'estimer les efforts dans la barre, un modèle d'actionneur dynamique n'a pas d'intérêt. De plus, les actionneurs utilisés à bord du trimaran MACIF sont des actionneurs électrohydrauliques, dont la vitesse de déplacement est relativement faible (environ 5 ° /s), mais peu dépendante de l'effort de barre. On utilise donc un modèle cinématique à vitesse constante : l'actionneur se déplace à une vitesse finie vers la consigne, et s'arrête quand il l'atteint.

Cette loi est implémentée dans un composant AMSA. Le template déclare simplement une entrée (la commande), une sortie (l'angle de barre) ainsi que deux paramètres : la vitesse angulaire et l'angle de butée. L'implémentation du modèle cinématique se fait dans la méthode *doStep()*, et nécessite l'ajout de 11 lignes de code à la classe C++ générée.

11.3.2 Le développement d'une nouvelle loi de pilotage

Lorsque l'on a connaissance de la nature de la loi que l'on veut intégrer (les équations ou l'algorithme qui la définit), il est possible soit de traduire ces équations dans le langage cible (le C++ dans notre cas), soit d'utiliser une implémentation existante que l'on appelle à chaque pas de temps. Pour illustrer cette possibilité et disposer d'une loi de contrôle facilement utilisable, nous utilisons ici une loi de type Proportionnelle - Dérivée - Dérivée (PDD) proposée par G. Guillou [Gui10]. Cette loi permet une régulation sur le cap du bateau, elle est décrite par l'équation suivante :

$$\theta = K_c(K_h(hdg - setPoint) + K_y \times yawRate + K_r \times rollRate)$$

Elle permet de calculer l'angle de barre θ en fonction du cap hdg, de la vitesse de rotation du bateau $yawRate$ et de la dérivée de l'angle de gîte $rollRate$, dans le but de minimiser l'écart entre le cap et la consigne $setPoint$. Elle comprend trois paramètres permettant de donner plus ou moins de poids aux contributions des termes proportionnel (K_h) et dérivés (K_y et K_r). Un paramètre K_c permet de régler le gain global de la loi. A l'image des pilotes existants, une limitation d'angle de barre maximum (et minimum pour les angles négatifs) est ajoutée. Cette valeur permet d'éviter que le contrôleur ne demande un angle de barre supérieur aux possibilités mécaniques du système.

La loi PDD porte sur la régulation du cap. Il est cependant courant de vouloir réguler l'angle que forme le bateau avec le vent (TWA). Une approche courante pour répondre à ce besoin est de calculer à chaque pas de temps une consigne de cap équivalente à l'angle au vent souhaité, et de garder la même loi de régulation, avec une consigne de cap qui suit les variations du vent. Le contrôleur dispose ainsi de deux modes de fonctionnement, le mode cap où l'utilisateur indique directement le cap qu'il désire suivre, et le mode vent où la consigne porte sur un angle au vent.

```
1  /*
2   * basicPilot.h
3   * Generated by Acceleo
4   */
5  #ifndef BASICPILOT_H_
6  #define BASICPILOT_H_
7  #include <string>
8
9  namespace basicPilot {
10
11     class AbstractBasicPilotInterface {
12     public:
13         virtual ~AbstractBasicPilotInterface(){}
14         virtual int setTarget(double target) = 0;
15         virtual int activateWindMode() = 0;
16         virtual int activateHeadingMode() = 0;
17     };
18 }
19
20 #endif /* BASICPILOT_H_ */
```

FIGURE 11.2 – Le *header* C++ généré à partir de l'interface *BasicPilot*

Le passage d'un mode à l'autre et les modifications de consigne nécessitent que l'utilisateur puisse envoyer des informations au pilote durant la simulation, par exemple par le biais du scénario. Cela est rendu possible par une interface de marquage *basicPilot*. La figure 11.2 montre le code C++ généré à partir de cette interface. En plus d'identifier le composant qui l'implémente en tant que pilote, l'interface propose deux méthodes de changement de mode (*activateWindMode* et *activateHeadingMode*) ainsi qu'une méthode de changement de consigne (*setTarget*).

Le template du composant *PddPilot* créé pour implémenter cette loi contient quatre entrées (*hdg, yawRate, rollRate* et *TWA*) et une sortie (la commande d'angle de barre *rudderCmd*). Elle fournit l'interface *basicPilot* et déclare cinq paramètres : les quatre de la loi de contrôle et la butée de barre. Comme cette loi est relativement simple et que l'on ne dispose pas d'une implémentation existante, on insère le code C++ directement dans la classe générée. La loi en elle-même tient sur 11 lignes dans la méthode *doStep()*. 6 lignes supplémentaires sont nécessaires pour gérer le changement de mode.

11.3.3 L'intégration d'un pilote existant

On souhaite à présent effectuer des simulations avec la loi de pilotage réellement embarquée sur le trimaran MACIF. Celui-ci est équipé d'un pilote automatique fortement paramétrable, développé par la société Madintec [Mad19]. Ce pilote est basé sur l'architecture matérielle présentée au paragraphe 2.3.3. Le calculateur (*MADBrain*) reçoit les informations en provenance de la centrale de navigation via un bus CAN, et envoie une consigne d'angle de barre sur ce même bus. La consigne est lue par une carte de puissance (*MADController*), connectée elle aussi au bus, qui commande l'actionneur. Un serveur web est embarqué dans le *MADBrain*, il propose une IHM qui permet de paramétrer le pilote via un navigateur depuis l'ensemble des terminaux connecté au réseau du bord.

Si le paramétrage de ce pilote est facile, la loi de contrôle utilisée par le pilote n'est ni accessible, ni connue. Le seul moyen d'intégrer cette loi de contrôle dans la simulation est donc de procéder à un montage de type *hardware in the loop*. Le boîtier du calculateur *MADBrain* est connecté au simulateur, celui-ci lui envoyant les données en lieu et place de la centrale de navigation et recevant en retour la consigne d'angle de barre. Ces communications s'effectuent via un bus CAN auquel l'ordinateur du simulateur est relié par un adaptateur CAN/USB. Une API *WebSocket* de configuration du pilote ayant été fournie par *Madintec*, l'ordinateur et le *MADBrain* sont également reliés via un réseau IP local, afin que le simulateur ait accès au paramétrage du pilote. Le montage ainsi obtenu est visible sur la figure 11.3

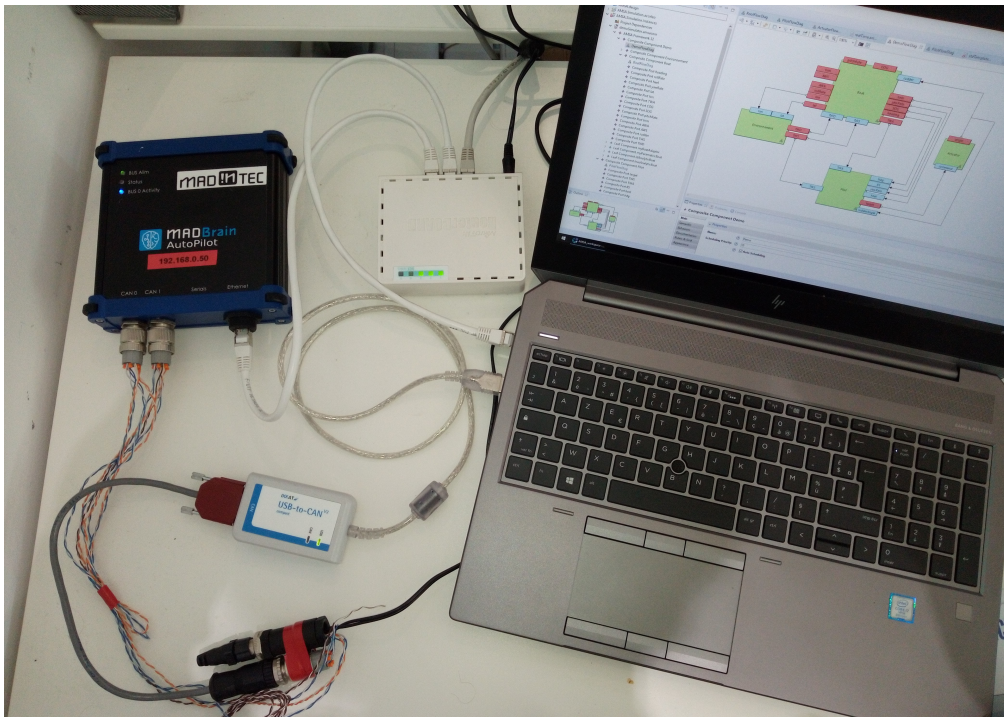

FIGURE 11.3 – Montage *hardware in the loop* du *MADBrain*

Un composant *MadBrainWrapper* est développé pour intégrer le calculateur pilote dans le simulateur. Celui-ci est en charge d'assurer la communication avec le *MADBrain* via le bus CAN. Dans ce but, il s'appuie sur une bibliothèque existante pour envoyer et recevoir des informations sur le bus. Les données sont envoyées sur le bus à chaque exécution de la méthode *doStep()*. Les données en provenance du bus (la consigne d'angle de barre) doivent être lues via un *thread* dédié. A son initialisation, le composant lance un *thread* d'écoute en utilisant les mécanismes offerts par la bibliothèque standard C++. Ce *thread* lit les données depuis le bus et les stocke dans un *buffer* interne au composant. La méthode *doStep()* est ensuite en charge de publier le contenu de ce *buffer* sur la sortie du composant. Le *thread* d'écoute est stoppé lors de l'appel à la méthode *finalize()* du composant. La gestion de ce thread et les appels à la bibliothèque permettant l'écriture de données demandent l'ajout d'une quarantaine de lignes de code au template généré.

Le composant *BrainWrapper* est également en charge de la configuration du *MADBrain* via l'API *WebSocket*. Pour cela, le template fournit l'interface de marquage *MadPilot*, qui contient des méthodes permettant d'activer les différents modes de pilotage, d'éditer la consigne et de modifier les paramètres de la loi de contrôle. Ces paramètres ne sont pas listés ici pour des raisons de confidentialité. Les appels à l'API *WebSocket* sont réalisés par une classe C++ développée pour l'occasion. Ainsi les fonctions de l'interface *MadPilot* sont de simples *wrappers* des méthodes de cette classe.

Comme l'échange d'informations et de paramètres avec le *MADBrain* s'effectue en temps réel, il est nécessaire que le framework utilise une horloge temps réel pour mener la simulation. Ce choix est effectué lors de l'édition du scénario de simulation. Si le contenu de celui-ci est identique au contenu d'un scénario exécuté en temps simulé, l'ajout du mot-clé *inRealTime* après le nom du scénario permet d'instancier l'horloge temps réel lors de la génération de code.

11.4 Bilan

Le développement des composants décrit dans ce chapitre permet d'évaluer les efforts nécessaires pour intégrer des modèles hétérogènes au sein du framework. Le tableau 11.1 récapitule les données nécessaires à l'estimation de ces efforts.

Nature	**Composant**	**Origine du modèle**	**Intégration du code**	**Lignes ajoutées**
Vent	ConstantWind	Dév. spécifique	Dans le template	3
	FileWindReader	Dév. spécifique	Dans le template	9
	WindPattern	Dév. spécifique	Dans le template	53
Bateau	AnalyticBoat	Proposé par [Jau04]	Dans le template	46
	ParametricBoat	Proposé par [Lav+16]	Classe C++ séparée	29
Actionneur	LinearRam	Dév. spécifique	Dans le template	11
Pilote	PddPilot	Proposé par [Gui10]	Dans le template	17
	MadBrainWrapper	Pilote Madintec	Bibliothèque externe	26

TABLE 11.1 – Les composants ajoutés au framework

On constate par le nombre de lignes de codes ajoutées que l'effort d'intégration est faible, grâce à la combinaison des mécanismes de génération de code et du pattern d'adaptation, qui permet d'exprimer les modèles dans leurs unités et dans leurs référentiels naturels.

Chapitre 12

Mise au point des lois de contrôle

Maintenant que nous disposons d'une bibliothèque de composants proposant des modèles de bateau, d'environnement ainsi que des lois de contrôle, il est possible de mener des simulations. Ce dernier chapitre présente la méthodologie utilisée pour mener ces simulations à travers l'élaboration de scénarios, l'obtention et l'analyse des résultats.

12.1 Méthodologie

12.1.1 La démarche

Les mécanismes de balayage proposés par les scénarios peuvent être utilisés pour trouver les valeurs optimales des paramètres d'une loi de pilotage dans une situation donnée. Afin de converger rapidement vers ces valeurs, la démarche suivante est adoptée pour chaque simulation :

1. Écriture d'un scénario comprenant des variations de conditions environnementales. Il est également possible d'ajouter des changements de la consigne du pilote automatique si l'on souhaite étudier son comportement dynamique durant les phases transitoires (ce qui correspond à l'approche classique en automatique d'étude de la réponse à un échelon).
2. Dans l'initialisation du scénario, initialisation du paramètre à étudier, avec un balayage portant sur un intervalle large, pour une dizaine de valeurs.
3. Ajout de deux observateurs : un premier pour créer un fichier de séries temporelles au format *Adrena* (que l'on nomme une trace *Adrena*) comprenant les données de navigation (vent, vitesse, position et attitude du bateau, angle de barre), et un second pour calculer un critère de performance dont la nature dépend de l'objectif de l'optimisation (qui peut être par exemple la maximisation de la vitesse du bateau, la minimisation de la distance parcourue ou de l'énergie utilisée par l'actionneur).
4. Après l'exécution du scénario, détermination des trois meilleures simulations selon le critère de performance, à l'aide du rapport généré par la simulation, puis visualisation des traces *Adrena* correspondantes. En effet, si le critère numérique fournit une idée globale de la performance, il est difficile de remplacer le ressenti de l'expert. Les navigateurs sont habitués à visualiser les informations sous forme de traces *Adrena*, ils peuvent comparer la qualité et la performance du pilotage par visualisation des différentes séries temporelles.
5. Si besoin, retour à l'étape 2 en réduisant l'intervalle de balayage autour du premier résultat obtenu, dans le but d'affiner la valeur finale.

Le choix du critère de performance dépend de l'objectif de la simulation. Plus celui-ci est pertinent, plus il est facile d'automatiser la recherche de l'optimum. Si l'utilisateur juge son critère suffisamment fiable pour éviter l'étape de vérification « à la main » des traces obtenues, l'automatisation permet d'envisager le balayage de plusieurs paramètres.

Il est également possible de s'intéresser à une optimisation multi-objectifs. Plusieurs critères doivent alors être déclarés dans le scénario. Ces critères seront présents dans le rapport de simulation, mais le framework ne fournit pas encore d'outils pour une analyse multicritères. Il est

actuellement nécessaire d'utiliser des outils externes pour tirer des conclusions sur les résultats de la simulation.

12.1.2 Élaboration d'un scénario de vent

La première étape de la démarche consiste à choisir un scénario de vent. Nous proposons ici un scénario typique, d'une durée de 120 secondes, qui sera le support des différentes simulations. Ce scénario contient trois phénomènes typiques : une rafale, une rotation à droite et un vent forcissant. Il est donc possible d'avoir recours au composant *WindPattern* pour générer le vent correspondant à ces trois phénomènes. La figure 12.1 montre le paramétrage de ce composant au sein du scénario de simulation, ainsi qu'une représentation graphique de la vitesse et de la direction du vent ainsi obtenu.

```
at 0 onEnvironment Demo.Environment.myWindGenerator
    WindPatternInterface.constantMode(20.0, 0.0);
at 30 onEnvironment Demo.Environment.myWindGenerator
    WindPatternInterface.rampRise(5, 15, 2.5);
at 60 onEnvironment Demo.Environment.myWindGenerator
    WindPatternInterface.turnRight(25, 5.0);
at 0 onEnvironment Demo.Environment.myWindGenerator
    WindPatternInterface.ramp(0.3, 30);
at 120 Stop
```

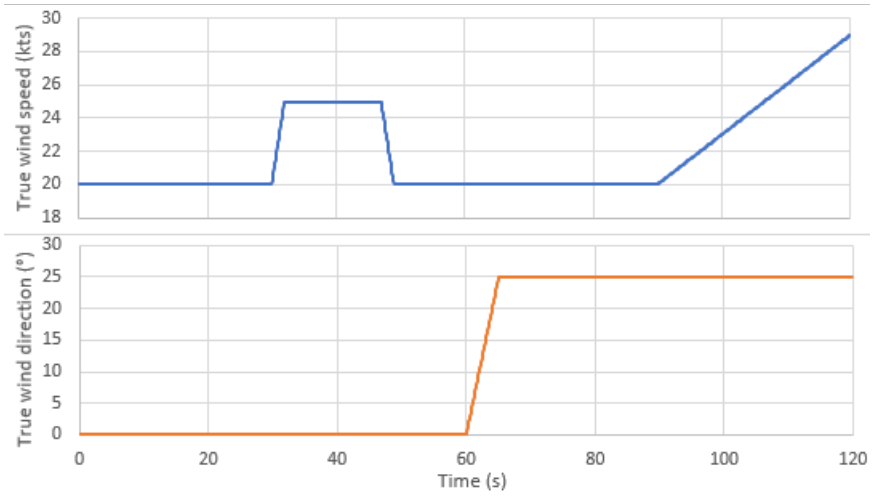

FIGURE 12.1 – Un scénario de vent utilisant la *WindPatternInterface*, et la représentation graphique du vent ainsi obtenu

Le scénario de vent considéré débute par un vent de nord (direction de 0°) constant d'une vitesse de 20 kts. Après 30 secondes, une rafale de +5 kts a lieu durant 15 secondes, avec une vitesse de montée et de descente de 2.5 kts/s. A 60 secondes, la direction du vent tourne vers la droite à une vitesse de 5°/s, pour une rotation totale de 25°. Après 90 secondes, la vitesse du vent augmente de 0.3 kts/s jusqu'à la fin de la simulation.

12.2 Simulations

A présent que la méthodologie de mise au point des lois de contrôle a été exposée, cette partie en présente les différentes applications à travers des simulations impliquant diverses lois de contrôle. Dans chaque cas, l'objectif de la simulation est exposé, les composants utilisés dans le simulateur sont listés, puis les résultats sont présentés et interprétés.

12.2.1 Validation des possibilités du framework

L'objectif de la première simulation est de tester les possibilités des balayages de scénarios en mettant au point la loi de contrôle PDD pour l'adapter au modèle analytique de voilier présenté au paragraphe 11.2.1. Il est important de remarquer que l'on souhaite utiliser un modèle de bateau

qui ne calcule pas d'angle de gîte (le seul DDL de rotation pris en compte est le cap) avec une loi qui utilise la dérivée de cet angle de gîte. Il est cependant possible de réaliser une simulation sans modification aucune de la loi ni du modèle de bateau. En effet, celui-ci ayant indiqué à l'adaptateur universel qu'il ne travaillait pas avec cet angle de gîte, l'adaptateur fournit une valeur par défaut, à savoir 0. Ainsi, le coefficient K_r de la loi de contrôle n'aura ici aucune influence.

Le comportement principal de la loi de contrôle ainsi réduite va dépendre du rapport entre le coefficient proportionnel K_h et le coefficient dérivé K_y, le gain global K_c venant simplement modifier l'amplitude des mouvements de barre. Pour faire évoluer ce rapport, on fixe K_h à une valeur arbitraire de 7, et on balaye K_y entre 0 et 10. Le fait de décentrer K_h en partie haute de l'intervalle résulte du fait que, pour ce type de loi, la contribution du terme proportionnel est souvent plus importante que la contribution du terme dérivé.

Le critère de performance retenu est la vitesse moyenne du voilier sur l'ensemble de la simulation. Le scénario de vent utilisé est celui décrit au paragraphe 12.1.2, ce qui implique une durée de simulation de 120 s. La régulation porte sur le cap, le pilote est donc activé en mode cap. Trois perturbations de la consigne sont ajoutées au scénario afin d'évaluer la robustesse de la loi de contrôle :

- la consigne initiale est de 120 ° alors que le cap initial du bateau est de 90 ° ;
- à $t = 40s$, la consigne est changée pour 60 ° ;
- à $t = 80s$, la consigne est remise à 120 ° .

Notons que pour réaliser la première perturbation, il est nécessaire de pouvoir choisir les conditions initiales du voilier. Le composant *BoatAdaptator* dispose à cet effet de quatre paramètres (*initialLat, initialLon, initialHdg* et *initialSpeed*) qui permettent, lors de l'initialisation du scénario, d'indiquer la position, la direction et la vitesse initiale du bateau. Lors de l'initialisation des composants, l'adaptateur réalise une transaction avec le modèle de bateau auquel il est relié. Si le modèle accepte une initialisation externe (ce qui est le cas pour le modèle analytique), l'adaptateur lui transmet les valeurs de ses paramètres, après les avoir éventuellement adaptées. Ce mécanisme permet une initialisation indépendante du modèle de bateau utilisé, en particulier en termes d'unité et de référentiel.

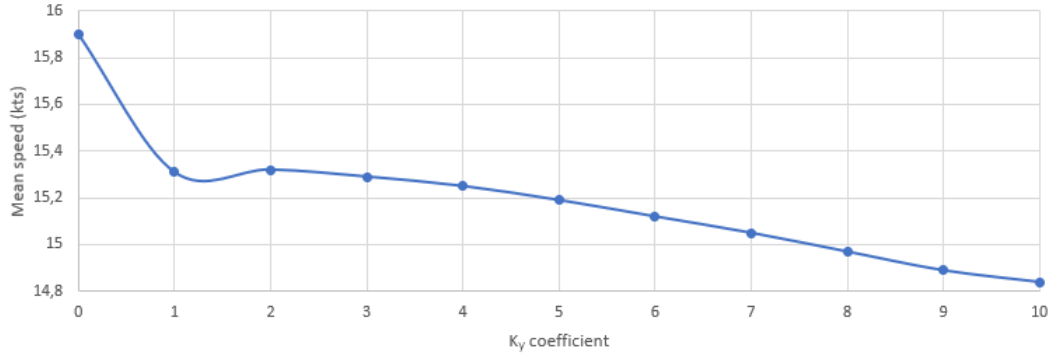

FIGURE 12.2 – Évolution de la vitesse moyenne en fonction de K_y

L'exécution de scénario comprend 11 simulations (une par valeur de K_y), qui s'effectuent en temps simulé. Sur un poste de travail, l'ensemble de ces simulations nécessite entre 3 et 4 secondes de calcul (comprenant l'écriture des fichiers de résultats sur le disque). La figure 12.2 présente l'évolution du critère de performance en fonction de la valeur du paramètre K_y. On y constate que la vitesse est maximum pour $K_y = 0$, puis décroit quand K_y augmente. Le critère donne donc la valeur 0 comme optimale. En suivant la procédure présentée ci-dessus, on visualise alors les différentes simulations à l'aide du logiciel *Adrena*, qui permet une représentation graphique des trajectoires, visible sur la figure 12.3. Cette représentation nous permet de faire deux observations. La première concerne la trajectoire correspondant à $K_y = 0$, qui est est instable : comme le terme dérivé est absent, le pilote ne parvient pas à anticiper les rotations du bateau, mais celui-ci conserve une vitesse élevée. La meilleure solution indiquée par le critère de performance n'est donc pas viable. La seconde observation porte sur le caractère oscillatoire des trajectoires : plus K_y est élevé, plus les oscillations sont amorties, mais moins le pilote est réactif (les virages interviennent plus tardivement, avec un rayon de courbure plus important). La trajectoire la plus réactive qui ne présente pas d'oscillations marquées correspond à $K_y = 2$. C'est également la configuration qui donne la vitesse moyenne la plus élevée, si l'on exclut la trajectoire instable.

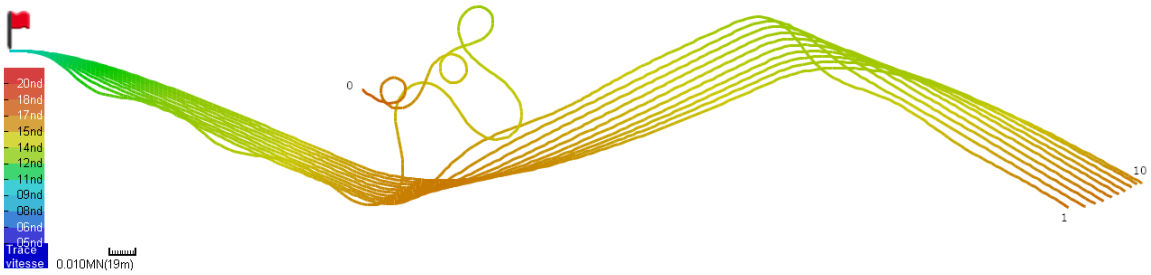

FIGURE 12.3 – Trajectoires des 11 simulations balayées. Tous les bateaux sont lancés au drapeau rouge. La valeur de K_y est indiquée en fin de trajectoire (les extrémités sont dans l'ordre entre 1 et 10). Les traces sont colorées en fonction de la vitesse des bateaux.

L'exploration paramétrique permet ainsi de déterminer la valeur optimale d'un coefficient de la loi de contrôle dans une situation donnée. Cette optimisation est menée selon un critère donné (ici la vitesse), ce qui ne garantit pas forcément la cohérence des résultats. L'expertise humaine des traces obtenues reste un moyen sûr de vérifier la pertinence des résultats de la simulation. Notons que dans ce cas, on cherche à mettre au point une loi de contrôle à l'aide d'un modèle de bateau dont les coefficients ont été déterminés de manière empirique. Il est donc impossible d'affirmer que les résultats obtenus sont transposables à un bateau réel. En effet, le modèle de bateau utilisé ne reproduit pas le comportement d'un bateau réel, mais un comportement qui « ressemble » à celui d'un bateau. Sans moyen de comparaison avec une situation réelle, il est difficile de déterminer qui, de la loi de contrôle ou du modèle de bateau, doit être mis au point. Par la suite, on a recours aux modèles paramétriques afin d'éviter ce problème.

12.2.2 Rejeu du vent et validation du modèle paramétrique

L'objectif de cette seconde simulation est de mettre en œuvre un modèle paramétrique afin de valider sa capacité à reproduire le comportement du trimaran MACIF. On utilise comme modèle d'environnement un enregistrement du vent réalisé lors d'une navigation d'entraînement. La série temporelle d'une durée de 220 secondes est donc rejouée par le composant *FileWindReader* afin de reproduire en simulation les conditions rencontrées durant la navigation. Le vent obtenu est assez instable, sa vitesse oscille entre 18 et 20 nœuds et sa direction entre 100 ° et 140 ° . On choisit donc un modèle paramétrique dont le domaine de validité est compatible avec la vitesse du vent, et l'orientation du bateau sera choisie pour que le TWA soit également dans ce domaine de validité.

Comme dans la première simulation, la loi de pilotage PDD est utilisée, mais c'est à présent le coefficient K_c (le gain global de la loi) qui est balayé : deux valeurs sont testées, respectivement 0.1 et 0.05. L'objectif est ici de tester la sensibilité du modèle paramétrique à l'amplitude de l'angle de barre.

Comme cela a été précisé au chapitre précédent, l'utilisation d'un modèle paramétrique implique que la simulation reste dans le domaine de validité du modèle, en particulier en termes d'allure (la valeur du TWA ne devant pas varier de plus de 10 ° à 30 ° suivant les modèles). Pour respecter cette condition, le pilote est placé en mode vent réel, la régulation porte donc sur le TWA, avec une consigne fixée à 120 ° .

Comme la loi de régulation a pour consigne de conserver un angle constant avec le vent, le critère de performance utilisé ici correspond à la moyenne de l'écart entre le TWA et la consigne du pilote. Il caractérise la capacité de la loi de contrôle à respecter la consigne. Les valeurs de critère obtenues par la simulation sont 3.78 ° et 2.23 ° , respectivement pour $k_c = 0.1$ et $k_c = 0.05$.

Les trajectoires obtenues sont visibles sur la figure 12.4. Celle-ci présente par ailleurs l'interface du logiciel Adrena, qui peut être paramétré pour suivre l'évolution des deux bateaux de manière simultanée (le bateau 1 correspond à $k_c = 0.1$ et le 2 à $k_c = 0.05$). On constate que les trajectoires sont proches, et qu'elles s'incurvent pour suivre les évolutions du vent (celui-ci étant représenté par les barbules le long des traces).

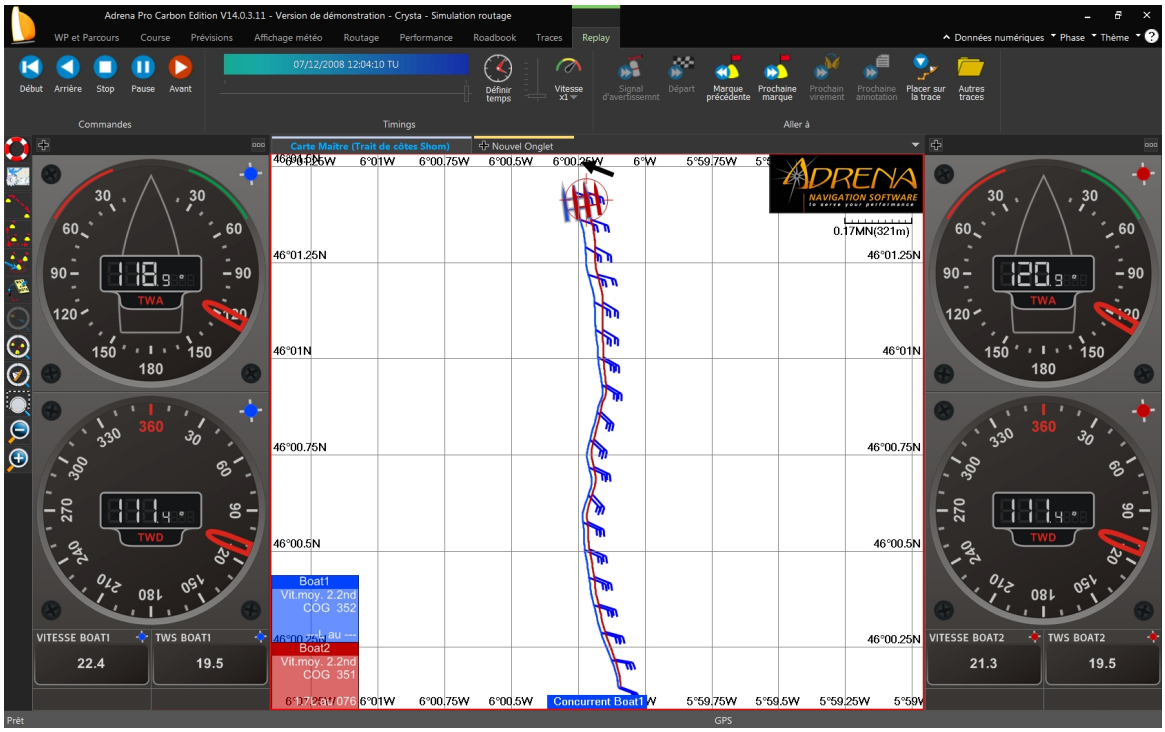

FIGURE 12.4 – Trajectoire des deux simulations. Les cadrans de gauche affichent les informations relatives au bateau 1 (bleu), ceux de droite sont dédiés au bateau 2 (rouge).

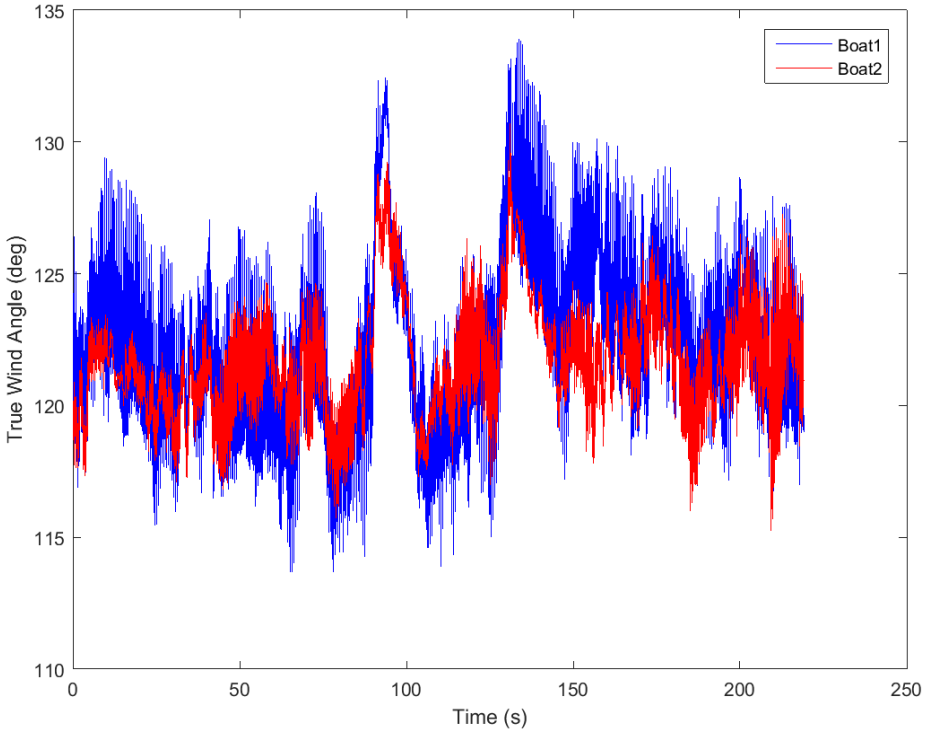

FIGURE 12.5 – Évolution du TWA durant la simulation

Pour comprendre les différence de comportement entre les deux bateaux, on représente les séries temporelles du TWA obtenu pour les deux bateaux (figure 12.5). On constate que le bateau 1 ($K_c = 0.5$) présente des oscillations plus marquées que le deuxième. En effet, une grande amplitude de barre entraîne une instabilité importante, même si cela permet d'avoir en moyenne un TWA plus proche de la consigne comme l'indique le critère numérique. Ces oscillation impactent peu la trajectoire globale du bateau, ce qui peut venir du fait que le modèle paramétrique estime mal la répercussions de tels phénomènes sur la performance du bateau. Un voilier réel présentant des oscillations à l'image du bateau 1 aura probablement une vitesse inférieure à un voilier ayant le comportement plus lisse du voilier 2. Malgré ces limitations, le modèle paramétrique permet tout de même d'obtenir des résultats sur le comportement du bateau : il réagit à la loi de pilotage et présente un comportement crédible en termes de vitesse moyenne et de direction.

12.2.3 Paramétrage du pilote *Madintec*

On cherche à présent à simuler le comportement du trimaran MACIF à l'aide des modèles paramétriques, dans l'objectif d'optimiser la valeur d'un coefficient K du pilote *Madintec*. Ce coefficient, qui correspond à un gain global, agit directement sur l'amplitude des corrections apportées par le pilote. Il peut varier entre les valeurs 0.5 et 1.5.

L'expérience des navigations révèle que ce coefficient a une influence assez marquée sur la nervosité du pilote (sa capacité à mettre des coups de barre). En revanche, l'influence sur la forme de la trajectoire est souvent moins nette. Afin de retirer un maximum d'informations des simulations, trois critères de performance différents seront calculés de manière simultanée : la vitesse moyenne, l'écart-type de l'angle de barre, et l'écart moyen entre la consigne et la valeur régulée.

De même que pour la simulation précédente, le pilote *Madintec* est placé en mode vent réel, avec une consigne fixée à -120 ° (le moins indiquant ici une navigation bâbord amure) afin de rester dans le domaine de validité du modèle paramétrique. Le scénario de vent utilisé est celui décrit au paragraphe 12.1.2, ce qui implique une durée de simulation de 120 s. Le modèle paramétrique correspondant au contexte (allure portante, vent d'une vingtaine de nœuds) est sélectionné. La valeur du paramètre K est balayée entre 0.5 et 1.5 par incréments de 0.25, ce qui produit cinq simulations.

Le tableau 12.1 présente les résultats obtenus sur les trois critères. Pour compléter l'analyse, les traces correspondantes sont visibles sur la figure 12.6. Le changement de cap qui survient au milieu de la simulation est dû à la rotation du vent indiquée dans le scénario. On constate par ailleurs que les vitesses sont corrélées avec la force du vent. L'analyse des trajectoires n'apporte pas de réel élément de comparaison entre les différentes simulations, les critères sont plus parlants. En effet, si la vitesse moyenne est quasiment constante avec des variations inférieures à 0.9% (à l'exception du cas $K = 0.5$ où elle apparaît plus faible), on remarque que l'écart type de l'angle de barre augmente significativement (d'un facteur 2) avec K. Ce critère est une image de la nervosité du pilote et de sa tendance à donner des coups de barre. Un pilote nerveux consomme plus, les coups de barre excessifs ont également tendance à ralentir le bateau, bien que ce phénomène ne semble pas être reproduit par le modèle paramétrique. Il est donc naturel de vouloir minimiser ce critère.

Le dernier critère correspond à l'écart moyen entre la consigne et le TWA effectif. Il représente la capacité du contrôleur à respecter la consigne, donc à emmener le bateau à l'endroit désiré. On constate que ce critère admet un minimum pour $K = 1$. Sans surprise, les faibles valeurs de K sont plus loin de la consigne car le pilote manque de réactivité. Il est plus surprenant que les

K	**Vitesse moyenne (kts)**	**Écart-type de l'angle de barre (°)**	**Écart moyen à la consigne (°)**
0.5	42.59	0.62	2.23
0.75	43.79	0.62	1.72
1.0	43.44	0.75	1.43
1.25	43.41	0.98	1.51
1.5	43.44	1.23	1.58

TABLE 12.1 – Les valeurs des trois critères obtenues par simulation

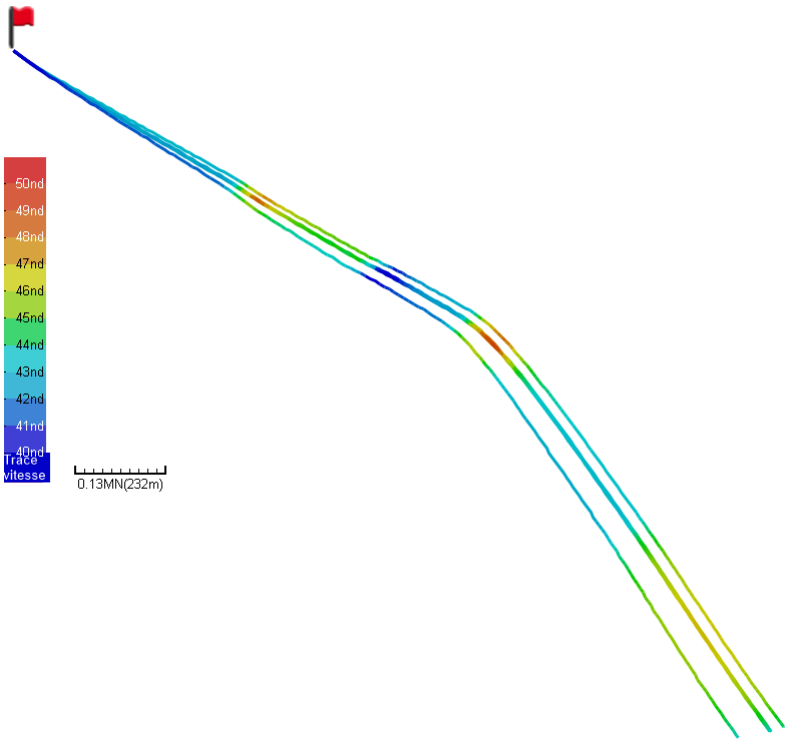

FIGURE 12.6 – Trajectoires des cinq simulations. Les traces de gauche et de droite correspondent respectivement aux valeurs de $K = 0.5$ et $Kk = 0.75$. Les autres traces sont superposées au centre.

valeurs élevées produisent des trajectoires éloignées de la consigne. Cela s'explique par l'apparition de petites oscillations qui, bien qu'elles ne soient pas visibles sur la trace, entraînent un tel écart.

Si l'on considère le troisième critère, la valeur $K = 1$ est optimale. L'écart type de l'angle de barre reste faible dans cette configuration, ce qui semble être le meilleur compromis, bien que la valeur $K = 0.75$ soit également intéressante du point de vue des deux premiers critères. La valeur de K utilisée sur le trimaran MACIF est de 1.0. Cette valeur a été déterminée de manière empirique au cours des différentes navigations qui ont servi à régler le pilote. La simulation permet ainsi de retrouver des résultats cohérents avec l'expérimentation.

12.3 Discussion sur la mise au point des lois de contrôle

Les possibilités offertes par le framework AMSA permettent de jouer des scénarios de simulations en automatisant le balayage de paramètres. Les critères de performance permettent une sélection automatique des meilleurs résultats, mais l'expertise reste nécessaire pour analyser les traces produites. Comme le framework ne propose pas encore d'outils ni pour la visualisation des critères ni pour la gestion de l'optimisation multi-objectifs, la phase d'interprétation des résultats reste de la responsabilité de l'utilisateur du framework.

Les simulations effectuées ont permis de retrouver des phénomènes connus et des valeurs précédemment déterminées de manière empirique. La démarche employée semble donc valide pour explorer de nouvelles possibilités de paramétrage, et il est envisageable de développer une nouvelle loi de contrôle à partir de ces outils. Cependant, la précision des résultats est fortement dépendante de la qualité des modèles de bateaux. Si les modèles utilisés dans cette étude sont suffisants pour obtenir des résultats globaux, ils trouvent rapidement leurs limites lorsque l'on souhaite les utiliser pour déterminer des paramètres plus fins qui interviennent sur des aspects subtils de la dynamique des voiliers de course, comme le départ au surf, le décollage ou encore le couplage gîte/dérive. Pour mettre au point ces paramètres par simulation, la méthode reste pertinente, mais il faudra disposer de modèles de voiliers plus précis et intégrer des modèles de mer.

Chapitre 13

Conclusion

13.1 Bilan

Ce travail s'intéresse aux problématiques de simulations de voiliers de compétition, et particulièrement aux aspects architecturaux, c'est-à-dire la facilité d'intégration de composants hétérogènes dans un simulateur et la facilité d'exploitation de ce dernier, à travers la conduite de simulations et l'exploitation de résultats. Nous dressons ici un bilan des différentes contributions qui ont été proposées en réponse à ces problématiques.

Pour répondre aux besoins architecturaux des logiciels de simulation, nous avons construit un environnement de modélisation spécifique au domaine visé, le framework AMSA, basé sur un modèle à composants qui permet de construire l'architecture d'un simulateur au moyen de langages spécifiques textuels et graphiques. A l'image des architectures matérielles présentes sur les voiliers, ce modèle est basé sur l'échange de flux de données. Des interfaces de communication viennent étendre ce paradigme, tandis qu'un mécanisme de typage par templates permet la définition et la réutilisation de composants métier.

Un style architectural a été proposé dans le but de répondre aux problèmes d'intégration de composants hétérogènes dans un simulateur. Ce style repose sur un pattern d'adaptation, dans lequel un adaptateur universel communique avec le composant à adapter au moyen d'une interface standardisée. Le travail de construction et de standardisation de ces interfaces a été mené pour les modèles d'environnement et les modèles de voilier, ce qui permet de réduire notablement l'effort d'intégration au simulateur d'un nouveau modèle de vent ou de bateau.

L'objectif de la simulation est de fournir des résultats permettant des avancées dans divers domaines (mise au point de lois de contrôle, architecture navale, ingénierie). Un simulateur doit ainsi être facilement exploitable. Pour répondre à ce besoin, nous avons proposé un mécanisme de scénario basé sur des événements, ce qui facilite le contrôle du déroulé d'une simulation, tout en automatisant certaines tâches comme l'exploration paramétrique ou la production de critères de comparaison des simulations. Les scénarios sont exprimés dans un langage spécifique dont la grammaire a été spécialement étudiée pour permettre la concision et la clarté.

Pour démontrer les possibilités d'intégration du framework AMSA, nous avons ensuite développé divers modèles de vent, de bateau et de pilote dans le framework afin de constituer une bibliothèque de composants métier. Les mécanismes de génération de code à partir des modèles ont permis de réduire considérablement l'effort d'intégration de ces modèles, en laissant au développeur le loisir de se consacrer entièrement au code métier.

Nous avons enfin validé les multiples possibilités offertes par le framework, tout d'abord en mettant en œuvre des modèles simples pour tester une loi de régulation classique, ce qui nous a permis de retrouver des résultats connus du domaine, puis en appliquant la démarche à un pilote du marché, à l'aide d'un modèle de bateau élaboré à partir de logs du trimaran MACIF. Les résultats des simulations ont permis de confirmer les réglages de ce pilote qui avaient été déterminés empiriquement lors de navigations.

13.2 Perspectives

Le framework AMSA offre déjà des possibilités intéressantes, et de nombreuses perspectives futures sont envisagées, tant du côté *MerConcept* que du côté Lab-STICC. Nous présentons ici les perspectives d'amélioration et d'évolution du framework et nous détaillons brièvement les enjeux que cela représente.

Vers une IHM dédiée

L'utilisation du framework AMSA repose actuellement entièrement sur les éditeurs de modèles présentés dans le chapitre 10. Si ces éditeurs ont été conçus pour faciliter l'édition de modèles et la génération de code, la compilation et l'exécution du code généré doit se faire « à la main », par exemple via un environnement de développement adapté au langage cible, ici le C++. L'utilisation d'Eclipse à la fois comme environnement de modélisation et comme environnement de développement permet de simplifier la démarche, mais cela reste relativement rebutant pour un utilisateur non coutumier des techniques de programmation. De plus, les rapports de simulations étant fournis sous forme de fichiers textuels, leur interprétation doit être faite manuellement.

Pour améliorer l'ergonomie du framework, il serait possible de construire une Interface Homme-Machine (IHM) permettant l'édition des modèles, la génération et la compilation du code et le lancement de l'exécution des simulations. Une telle interface pourrait également inclure des outils de visualisation des résultats, comme un affichage graphique des critères de performances ou encore des outils spécifiques à la gestion du multi-objectifs. Une partie de cette IMH pourrait être dédiée à la simulation temps réel, en proposant une fenêtre de commande du pilote identique à celle présente sur les bateaux, et une zone de visualisation de la trajectoire du bateau en temps réel. Une telle configuration autoriserait alors l'utilisateur à agir en direct sur le pilote, comme il le ferait en situation réelle.

Vers des modèles de bateaux plus performants

Ce travail n'était pas centré sur la précision des modèles de bateaux, à savoir leur capacité à reproduire le comportement réel d'un bateau. Cependant, si les modèles paramétriques sont suffisants pour obtenir des résultats qui vont « dans le bon sens », ils ne permettent pas de capturer toute la subtilité des réactions d'un engin de course comme le trimaran MACIF. En particulier, les problématiques de décollage, de stabilité en vol ou de sensibilité à l'état de mer sont totalement absentes des modèles paramétriques. La mise au point et le réglage de lois de contrôle s'intéressant à ces problématiques nécessitent ainsi de disposer de modèles plus précis.

La construction de tels modèles représente un sujet d'étude à part entière, auquel s'intéressent de près les architectes : un outil permettant de prévoir la stabilité et la performance constitue un véritable atout pour la conception. De tels outils font leur apparition, souvent désignés sous le terme de *VPP dynamique* ou plus simplement *simulateur*. Ils sont basés sur une étude approfondie des phénomènes aérodynamiques et hydrodynamiques qui entourent le bateau, ce qui nécessite une expertise couplée à une importante puissance de calcul. Les modèles présentés dans cette thèse ne prétendent pas rivaliser avec de tels outils, mais le framework prévoit les mécanismes nécessaires pour les intégrer au sein de simulateurs s'ils deviennent disponibles. Cela permettrait l'obtention de résultats beaucoup plus pertinents sur la mise au point des contrôleurs.

Une autre voie à explorer serait l'amélioration des modèles existants ou la mise au point de nouveaux modèles à partir d'études statistiques des séries temporelles (*Data Mining*) permettant d'identifier des motifs récurrents et des corrélations dans les données enregistrées lors de navigations.

Vers des lois de pilotage évoluées

Les lois de pilotage sont perçues dans ce travail comme des ensembles monolithiques paramétrables. Cependant, une approche multi-agents proposée par G. Guillou [Gui10] considère le pilote automatique comme une composition de plusieurs agents (réactif, adaptatif et expert) échangeant des données et des messages afin d'assurer un pilotage de qualité. Le système hiérarchique du framework AMSA pourrait permettre la modélisation d'une telle architecture : les différents agents

seraient autant de composants qui communiquent entre eux via des interfaces de service, tous étant contenus dans un composant composite qui apparaîtrait depuis l'extérieur comme un pilote monolithique classique.

Une telle approche, combinée aux possibilités offertes par AMSA, permettrait la mise au point agent par agent d'un nouveau de pilote, entièrement modélisé au sein du framework. Pour respecter le style architectural d'AMSA, une telle modélisation nécessiterait cependant une extension du framework avec l'introduction des ports d'interfaces composites, qui, à l'instar des ports de données composites, permettraient aux liens d'interfaces de « traverser » les frontières des composants composites.

Vers une utilisation embarquée

La capacité du framework AMSA à intégrer de nouveaux composants, la mise à disposition de modèles légers qui autorisent une simulation « plus rapide que le temps réel » et l'indépendance de toute plateforme matérielle du code généré permettent d'imaginer la possibilité d'embarquer un simulateur à bord du voilier, dans l'objectif de faire tourner des simulations en intégrant les conditions environnementales rencontrées. Il deviendrait alors possible de reconfigurer automatiquement le pilote en fonction des résultats des simulations. Un simulateur embarqué autoriserait également la mise en place d'algorithmes d'apprentissage qui viendraient enrichir les modèles du bateau au fil des navigations.

Vers d'autres types de contrôle

Le framework AMSA a été développé spécifiquement pour la mise au point de lois de contrôle sur les bateaux de course. Comme le seul asservissement autorisé concerne le pilote automatique de lacet, l'ensemble de ce travail concerne les lois de contrôle de l'angle de barre. Cependant, il n'est pas exclu que, dans les années à venir, d'autres appendices puissent être asservis : deuxième degré de liberté sur les safrans, incidence des foils, plans porteurs sur les dérives... Les possibilités sont nombreuses si l'on cherche à stabiliser le vol de tels engins. A cet effet, les outils de modélisation et le style architectural proposés dans ce travail pourraient être adaptés pour permettre des simulations autour de ces nouveaux objectifs.

Vers d'autres domaines d'application

La plupart des problématiques autour de la simulation rencontrées dans ce travail ne sont pas spécifiques au domaine de la voile. On les retrouve dans de nombreux cas où l'on souhaite simuler le comportement d'un système cyber-physique en interaction avec son environnement, comme par exemple pour les drones aériens ou sous-marins. Les outils du framework sont suffisamment génériques pour être adaptés à de nouvelles problématiques, au prix d'une redéfinition partielle du style architectural et surtout des interfaces standardisées.

Bibliographie

[Add+17] Lorenzo Addazi, Federico Ciccozzi, Philip Langer et Ernesto Posse. "Towards Seamless Hybrid Graphical–Textual Modelling for UML and Profiles". In : *European Conference on Modelling Foundations and Applications*. Springer. 2017, p. 20-33.

[Adr18] Adrena. *Adrena - Navigation Software to serve your performance*. 2018. URL : https://www.adrena-software.com/.

[And+05] Noriaki Ando, Takashi Suehiro, Kosei Kitagaki, Tetsuo Kotoku et Woo-Keun Yoon. "RT-middleware : distributed component middleware for RT (robot technology)". In : *Intelligent Robots and Systems, 2005.(IROS 2005). 2005 IEEE/RSJ International Conference on*. IEEE. 2005, p. 3933-3938.

[Ari+10] Soraya Arias, Florine Boudin, Roger Pissard-Gibollet et Daniel Simon. "Orccad, robot controller model and its support using eclipse modeling tools". In : *5th National Conference on Control Architecture of Robots*. 2010.

[BBS06] Ananda Basu, Marius Bozga et Joseph Sifakis. "Modeling heterogeneous real-time components in BIP". In : *Software Engineering and Formal Methods, 2006. SEFM 2006. Fourth IEEE International Conference on*. IEEE. 2006, p. 3-12.

[BCM12] Marco Brambilla, Jordi Cabot et Wimmer Manuel. *Model-Driven Software Engineering in Practice*. Synthesis Lectures on Software Engineering. Morgan & Claypool, 2012.

[BD07] Jean-Philippe Babau et Julien DeAntoni. "Architectures logicielles pour les systèmes embarqués temps réel". In : *Ecole d'été temps réel*. 2007.

[Ber04] Jean-Yves Bernot. *Météo et stratégie : croisière et course au large*. Fédération française de voile. Gallimard, 2004.

[Beu+99] Antoine Beugnard, Jean-Marc Jézéquel, Noël Plouzeau et Damien Watkins. "Making components contract aware". In : *Computer* 32.7 (1999), p. 38-45.

[BG11] Radhakisan Baheti et Helen Gill. "Cyber-physical systems". In : *The impact of control technology* 12.1 (2011), p. 161-166.

[BGM02] Marius Bozga, Susanne Graf et Laurent Mounier. "IF-2.0 : A validation environment for component-based real-time systems". In : *International Conference on Computer Aided Verification*. Springer. 2002, p. 343-348.

[Bil+96] Greet Bilsen, Marc Engels, Rudy Lauwereins et Jean Peperstraete. "Cyclo-static dataflow". In : *IEEE Transactions on signal processing* 44.2 (1996), p. 397-408.

[Bin+08] Jonathan R Binns, Karsten Hochkirch, Frank De Bord et Ian A Burns. "The development and use of sailing simulation for IACC starting manoeuvre training". In : *3rd High Performance Yacht Design Conference* (2008).

[Blo+11] Torsten Blochwitz, Martin Otter, Martin Arnold, Constanze Bausch, H Elmqvist, A Junghanns, J Mauss, M Monteiro, T Neidhold, Dietmar Neumerkel et al. "The functional mockup interface for tool independent exchange of simulation models". In : *Proceedings of the 8th International Modelica Conference ; March 20th-22nd ; Technical University ; Dresden ; Germany*. 063. Linköping University Electronic Press. 2011, p. 105-114.

[Bor+98] Jean-Jacques Borrelly, Éve Coste-Maniere, Bernard Espiau, Konstantinos Kapellos, Roger Pissard-Gibollet, Daniel Simon et Nicolas Turro. "The ORCCAD architecture". In : *The International Journal of Robotics Research* 17.4 (1998), p. 338-359.

[Bou+18] Youssef Bouanan, Simon Gorecki, Judicael Ribault, Gregory Zacharewicz et Nicolas Perry. "Including in HLA federation functional mockup units for supporting interoperability and reusability in distributed simulation". In : *Proceedings of the 50th Computer Simulation Conference*. Society for Computer Simulation International. 2018, p. 23.

[But+17] Arvid Butting, Arne Haber, Lars Hermerschmidt, Oliver Kautz, Bernhard Rumpe et Andreas Wortmann. "Systematic language extension mechanisms for the MontiArc architecture description language". In : *European Conference on Modelling Foundations and Applications*. Springer. 2017, p. 53-70.

[BW00] Eva Bauer et Ralf Weisse. "Determination of high-frequency wind variability from observations and application to North Atlantic wave modeling". In : *Journal of Geophysical Research : Oceans* 105.C11 (2000), p. 26179-26190.

[Cla14] Nicholas Adam Clark. "Validation of a sailing simulator using full scale experimental data". Thèse de doct. University of Tasmania, 2014.

[CSC08] François Christophe, Raivo Sell et Eric Coatanéa. "Conceptual design framework supported by dimensional analysis and system modelling language". In : *Estonian Journal of Engineering* 14.4 (2008), p. 303-317.

[Da 15] Alberto Rodrigues Da Silva. "Model-driven engineering : A survey supported by the unified conceptual model". In : *Computer Languages, Systems & Structures* 43 (2015), p. 139-155.

[Dah97] Judith S Dahmann. "High level architecture for simulation". In : *Proceedings First International Workshop on Distributed Interactive Simulation and Real Time Applications*. IEEE. 1997, p. 9-14.

[DB06] Julien DeAntoni et Jean-Philippe Babau. "Model driven engineering method for SAIA architecture design". In : *Ingénierie Dirigée par les Modèles*. 2006.

[DCL08] Zuohua Ding, Zhenbang Chen et Jing Liu. "A rigorous model of service component architecture". In : *Electronic Notes in Theoretical Computer Science* 207 (2008), p. 33-48.

[DeA07] Julien DeAntoni. "SAIA : Un style architectural pour assurer l'indépendance vis-à-vis d'entrées/sorties soumises à des contraintes temporelles". Thèse de doct. INSA de Lyon, 2007.

[Des+17] Henrick Deschamps, Gerlando Cappello, Janette Cardoso et Pierre Siron. "Toward a formalism to study the scheduling of cyber-physical systems simulations". In : *2017 IEEE/ACM 21st International Symposium on Distributed Simulation and Real Time Applications (DS-RT)*. IEEE. 2017, p. 1-8.

[DGL17] David Doose, Christophe Grand et Charles Lesire. "MAUVE Runtime : a component-based middleware to reconfigure software architectures in real-time". In : *2017 First IEEE International Conference on Robotic Computing (IRC)*. IEEE. 2017, p. 208-211.

[DGR09] Didier Donsez, Kiev Gama et Walter Rudametkin. "Developing adaptable components using dynamic languages". In : *35th Euromicro Conference on Software Engineering and Advanced Applications*. IEEE. 2009, p. 396-403.

[DM98] Judith S Dahmann et Katherine L Morse. "High level architecture for simulation : An update". In : *Proceedings. 2nd International Workshop on Distributed Interactive Simulation and Real-Time Applications (Cat. No. 98EX191)*. IEEE. 1998, p. 32-40.

[DTL15] Simone Di Cola, Cuong Tran et Kung-Kiu Lau. "A graphical tool for model-driven development using components and services". In : *2015 41st Euromicro Conference on Software Engineering and Advanced Applications (SEAA)*. IEEE. 2015, p. 181-182.

[ECB06] Jean-Paul Etienne, Julien Cordry et Samia Bouzefrane. "Applying the CBSE paradigm in the real time specification for Java". In : *Proceedings of the 4th international workshop on Java technologies for real-time and embedded systems*. ACM. 2006, p. 218-226.

[Ecl19a] Eclipse. *Acceleo - Generate anything from any EMF model*. 2019. URL : https://www.eclipse.org/acceleo/.

[Ecl19b] Eclipse. *Eclipse Modeling Framework, EMF*. 2019. URL : https://www.eclipse.org/modeling/emf/.

[Ecl19c] Eclipse. *Graphical Modeling Project (GMP)*. 2019. URL : https://www.eclipse.org/modeling/gmp/.

[Ecl19d] Eclipse. *Sirius - The easiest way to get your own Modeling Tool*. 2019. URL : https://www.eclipse.org/sirius/.

[Ecl19e] Eclipse. *Xtext - Language Engineering For Everyone !* 2019. URL : https://www.eclipse.org/Xtext/.

[Féd17] Fédération Française de Voile. *Les Règles de course à la voile 2017 - 2020*. 2017. URL : http://www.ffvoile.fr/ffv/web/ffvoile/documents/RCV/RCV_2017_2020/RCV_2017-2020.pdf.

[Fei+04] Peter H Feiler, Bruce Lewis, Steve Vestal et Ed Colbert. "An overview of the SAE architecture analysis & design language (AADL) standard : a basis for model-based architecture-driven embedded systems engineering". In : *IFIP World Computer Congress, TC 2*. Springer. 2004, p. 3-15.

[FMI18] FMI standard group. *Functional Mock-up Interface (FMI)*. 2018. URL : https://fmi-standard.org/.

[FR07] Robert France et Bernhard Rumpe. "Model-driven development of complex software : A research roadmap". In : *2007 Future of Software Engineering*. IEEE Computer Society. 2007, p. 37-54.

[Gam+95] Erich Gamma, Ralph Johnson, Richard Helm et John Vlissides. *Design Patterns*. Addison-Wesley, 1995.

[Gaz19] Gazebo. *GAZEBO - Robot simulation made easy*. 2019. URL : http://gazebosim.org/.

[GB13] Goulven Guillou et Jean-Philippe Babau. "IMOCA : une architecture à base de modes de fonctionnement pour une application de contrôle dans un environnement incertain". In : *CAL 2013. 7ième conférence francophone sur les architectures logicielles*. 2013.

[GB16] Goulven Guillou et Jean-Philippe Babau. "ImocaGen : A model-based code generator for embedded systems tuning". In : *2016 4th International Conference on Model-Driven Engineering and Software Development (MODELSWARD)*. IEEE. 2016, p. 390-396.

[Gen+02] Thomas Genssler, Alexander Christoph, Michael Winter, Oscar Nierstrasz, Stéphane Ducasse, Roel Wuyts, Gabriela Arévalo, Bastiaan Schönhage, Peter Müller et Chris Stich. "Components for embedded software : the PECOS approach". In : *Proceedings of the 2002 international conference on Compilers, architecture, and synthesis for embedded systems*. ACM. 2002, p. 19-26.

[GH06] Cesar Gonzalez-Perez et Brian Henderson-Sellers. "A powertype-based metamodelling framework". In : *Software & Systems Modeling* 5.1 (2006), p. 72-90.

[GLD14] Nicolas Gobillot, Charles Lesire et David Doose. "A modeling framework for software architecture specification and validation". In : *International Conference on Simulation, Modeling, and Programming for Autonomous Robots*. Springer. 2014, p. 303-314.

[Gre+15] Timo Greifenberg, Katrin Hölldobler, Carsten Kolassa, Markus Look, Pedram Mir Seyed Nazari, Klaus Mueller, Antonio Navarro Perez, Dimitri Plotnikov, Dirk Reiss, Alexander Roth et al. "A comparison of mechanisms for integrating handwritten and generated code for object-oriented programming languages". In : *2015 3rd International Conference on Model-Driven Engineering and Software Development (MODELSWARD)*. IEEE. 2015, p. 74-85.

[GS05] Gregor Gossler et Joseph Sifakis. "Composition for component-based modeling". In : *Science of Computer Programming* 55.EPFL-CONF-185043 (2005), p. 161-183.

[Gue+16] Sahar Guermazi, Saadia Dhouib, Arnaud Cuccuru, Camille Letavernier et Sébastien Gérard. "Integration of UML models in FMI-based co-simulation". In : *2016 Symposium on Theory of Modeling and Simulation (TMS-DEVS)*. IEEE. 2016, p. 1-8.

[Gui10] Goulven Guillou. "Architecture multi-agents pour le pilotage automatique des voiliers de compétition et extensions algébriques des réseaux de Petri". Thèse de doct. Université de Bretagne Occidentale, 2010.

[HDB17] Nicolas Hili, Juergen Dingel et Alain Beaulieu. “Modelling and code generation for real-time embedded systems with UML-RT and Papyrus-RT”. In : *2017 IEEE/ACM 39th International Conference on Software Engineering Companion (ICSE-C)*. IEEE. 2017, p. 509-510.

[Hei+11] Florian Heidenreich, Jendrik Johannes, Sven Karol, Mirko Seifert et Christian Wende. “Model-based language engineering with EMFText”. In : *International Summer School on Generative and Transformational Techniques in Software Engineering*. Springer. 2011, p. 322-345.

[HL95] Walter L. Hürsch et Cristina Videira Lopes. *Separation of Concerns*. Rapp. tech. College of Computer Science, Northeastern University, 1995.

[Hoš+10] Petr Hošek, Tomáš Pop, Tomáš Bureš, Petr Hnětynka et Michal Malohlava. “Comparison of component frameworks for real-time embedded systems”. In : *International Symposium on Component-Based Software Engineering*. Springer. 2010, p. 21-36.

[HRR14] Arne Haber, Jan Oliver Ringert et Bernhard Rumpe. “Montiarc-architectural modeling of interactive distributed and cyber-physical systems”. In : *arXiv preprint arXiv :1409.6578* (2014).

[IEE10] IEEE. *IEEE 1516-2010 - IEEE Standard for Modeling and Simulation (M&S) High Level Architecture (HLA) – Framework and Rules*. 2010. URL : https://standards.ieee.org/standard/1516-2010.html.

[Jau+12] Luc Jaulin, Fabrice Le Bars, Benoît Clement, Yvon Gallou, Olivier Menage, Olivier REYNET, Jan Sliwka et Benoît Zerr. “Suivi de route pour un robot voilier”. In : *Conférence Internationale Francophone d’Automatique (CIFA2012)*. Grenoble, France, 2012, p. 695-702.

[Jau04] Luc Jaulin. “Modélisation et commande d’un bateau à voile”. In : *CIFA’2004 (Conférence Internationale Francophone d’Automatique)*. 2004.

[Jen10] Fabian Jenne. “Simulation & Control Optimization of an autonomous sail boat”. In : *Swiss Federal Institute of Technology Zurich, Zürich* (2010).

[KH04] Nathan Koenig et Andrew Howard. “Design and use paradigms for gazebo, an open-source multi-robot simulator”. In : *2004 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)(IEEE Cat. No. 04CH37566)*. T. 3. IEEE. 2004, p. 2149-2154.

[Kru04] Philippe Kruchten. *The rational unified process : an introduction*. Addison-Wesley Professional, 2004.

[KRV10] Holger Krahn, Bernhard Rumpe et Steven Völkel. “MontiCore : a framework for compositional development of domain specific languages”. In : *International journal on software tools for technology transfer* 12.5 (2010), p. 353-372.

[Las+09] Gilles Lasnier, Bechir Zalila, Laurent Pautet et Jérome Hugues. “Ocarina : An environment for aadl models analysis and automatic code generation for high integrity applications”. In : *International Conference on Reliable Software Technologies*. Springer. 2009, p. 237-250.

[Lav+16] Emilien Lavigne, Benoit Piquemal, Adeline Bourdon, Simon Chesné, Goulven Guillou et Jean-Philippe Babau. “A process for evaluating parametric models for mechanical systems simulation : the case of a sailboat”. In : *Sofware and Hardware Architectures for Robots Control* (2016).

[LDC11] Charles Lesire, David Doose et Hugues Cassé. “Validation of real-time properties of a robotic software architecture”. In : *6th National Conference on Control Architectures of Robots*. 2011.

[LDG10] Gaëlle Lortal, Saadia Dhouib et Sébastien Gérard. “Integrating ontological domain knowledge into a robotic DSL”. In : *International Conference on Model Driven Engineering Languages and Systems*. Springer. 2010, p. 401-414.

[Lee15] Edward Lee. “The past, present and future of cyber-physical systems : A focus on models”. In : *Sensors* 15.3 (2015), p. 4837-4869.

[LLE07] Kung-Kiu Lau, Ling Ling et Perla Velasco Elizondo. “Towards composing software components in both design and deployment phases”. In : *International Symposium on Component-Based Software Engineering*. Springer. 2007, p. 274-282.

[Lom+12] Matteo Lombardi, Nicola Parolini, Alfio Quarteroni et Gianluigi Rozza. "Numerical simulation of sailing boats : Dynamics, FSI, and shape optimization". In : *Variational Analysis and Aerospace Engineering : Mathematical Challenges for Aerospace Design*. Springer, 2012, p. 339-377.

[LT12] Kung-Kiu Lau et Cuong M Tran. "X-MAN : An MDE tool for component-based system development". In : *2012 38th Euromicro Conference on Software Engineering and Advanced Applications*. IEEE. 2012, p. 158-165.

[Mad19] Madintec. *Madintec - electronics for foiling*. 2019. URL : http://madintec.com/.

[Mah13] Imran Mahmood. "A Verification Framework for Component Based Modeling and Simulation : Putting the pieces together". Thèse de doct. KTH Royal Institute of Technology, 2013.

[Man+16] Musa Morena Marcusso Manhães, Sebastian A Scherer, Martin Voss, Luiz Ricardo Douat et Thomas Rauschenbach. "UUV simulator : A gazebo-based package for underwater intervention and multi-robot simulation". In : *OCEANS 2016 MTS/IEEE Monterey*. IEEE. 2016, p. 1-8.

[Mas+14] Alejandro Masrur, Michal Kit, Tomá Bure et Wolfram Hardt. "Towards component-based design of safety-critical cyber-physical applications". In : *17th Euromicro Conference on Digital System Design*. IEEE. 2014, p. 254-261.

[Mat18] MathWorks. *MATLAB®, Simulink®*. 2018. URL : https://fr.mathworks.com/products/simulink.html.

[Mel15] Jon Melin. "Modeling, control and state-estimation for an autonomous sailboat". Mém. de mast. Uppsala University, 2015.

[MEO98] Sven Erik Mattsson, Hilding Elmqvist et Martin Otter. "Physical system modeling with Modelica". In : *Control Engineering Practice* 6.4 (1998), p. 501-510.

[Moo+09] J Mooney, N Saunders, M Habgood et JR Binns. "Multiple Applications of Sailing Simulation". In : *SimTecT*. T. 1. 2009, p. 489-494.

[MP08] Philippe Martinet et Bruno Patin. "Proteus : A platform to organise transfer inside french robotic community". In : *3rd National Conference on Control Architectures of Robots (CAR)*. 2008.

[Mro02] Zbigniew Mrozek. "Design of the mechatronic system with help of UML diagrams". In : *Proceedings of the Third International Workshop on Robot Motion and Control, 2002. RoMoCo'02.* IEEE. 2002, p. 243-248.

[Mro03] Zbigniew Mrozek. "Computer aided design of mechatronic systems". In : *International Journal of Applied Mathematics and Computer Science* 13.2 (2003), p. 255-267.

[MV13] Fabian A Mulder et Jouke C Verlinden. "Development of a motion system for an advanced sailing simulator". In : *Procedia Engineering* 60 (2013), p. 428-434.

[NBP13] Juan F Navas, Jean-Philippe Babau et Jacques Pulou. "Reconciling run-time evolution and resource-constrained embedded systems through a component-based development framework". In : *Science of Computer Programming* 78.8 (2013), p. 1073-1098.

[NCN17] S Nava, John Cater et Stuart Norris. "Large eddy simulation of downwind sailing". In : *INNOVSAIL International Conference on Innovation in High Performance Sailing Yachts, 4th Edition*. 2017, p. 127-138.

[Nes+06] Issa AD Nesnas, Reid Simmons, Daniel Gaines, Clayton Kunz, Antonio Diaz-Calderon, Tara Estlin, Richard Madison, John Guineau, Michael McHenry, I-Hsiang Shu et al. "CLARAty : Challenges and steps toward reusable robotic software". In : *International Journal of Advanced Robotic Systems* 3.1 (2006), p. 5.

[Nic+02] Cristian Nichita, Dragos Luca, Brayima Dakyo et Emil Ceanga. "Large band simulation of the wind speed for real time wind turbine simulators". In : *IEEE Transactions on energy conversion* 17.4 (2002), p. 523-529.

[OMG08] OMG. *MOF Model to Text Transformation Language*. 2008. URL : https://www.omg.org/spec/MOFM2T.

[OMG14] OMG. *Object Constraint Language, OCL 2.4*. 2014. URL : https://www.omg.org/spec/OCL/.

[OMG16] OMG. *Meta Object Facility, MOF 2.5.1*. 2016. URL : https://www.omg.org/spec/MOF.

[OMG17a] OMG. *Unified Modeling Language, UML 1.5*. 2017. URL : https://www.omg.org/spec/SysML/.

[OMG17b] OMG. *Unified Modeling Language, UML 2.5.1*. 2017. URL : https://www.omg.org/spec/UML/.

[Par04] Marc PARENTHOËN. "Animation phénoménologique de la mer—une approche énactive—". Thèse de doct. Université de Bretagne occidentale-Brest, 2004.

[Pas+14] Robin PASSAMA, David ANDREU, Didier CRESTANI et Karen GODARY-DEJEAN. "Architectures de contrôle pour la robotique-approches et tendances". In : *Techniques de l'ingénieur*. Editions TI | Techniques de l'Ingénieur, 2014, #S7791.

[Ric13] Johnhenri R RICHARDSON. "Quantifying and Scaling Airplane Performance in Turbulence." Thèse de doct. University of Michigan, 2013.

[Rom+10] Daniel ROMERO, Romain ROUVOY, Lionel SEINTURIER et Frédéric LOIRET. "Integration of heterogeneous context resources in ubiquitous environments". In : *2010 36th EUROMICRO Conference on Software Engineering and Advanced Applications*. IEEE. 2010, p. 123-126.

[ROS19] ROS. *ROS - Robot Operating System*. 2019. URL : https://www.ros.org/.

[RRW14] Jan Oliver RINGERT, Bernhard RUMPE et Andreas WORTMANN. "MontiArcAutomaton : Modeling Architecture and Behavior of Robotic Systems". In : *arXiv preprint arXiv :1409.2310* (2014).

[San+16] Jose Luis SANCHEZ-LOPEZ, Ramón A Suárez FERNÁNDEZ, Hriday BAVLE, Carlos SAMPEDRO, Martin MOLINA, Jesus PESTANA et Pascual CAMPOY. "Aerostack : An architecture and open-source software framework for aerial robotics". In : *2016 International Conference on Unmanned Aircraft Systems (ICUAS)*. IEEE. 2016, p. 332-341.

[San+17] Jose Luis SANCHEZ-LOPEZ, Martin MOLINA, Hriday BAVLE, Carlos SAMPEDRO, Ramón A Suárez FERNÁNDEZ et Pascual CAMPOY. "A multi-layered component-based approach for the development of aerial robotic systems : the aerostack framework". In : *Journal of Intelligent & Robotic Systems* 88.2-4 (2017), p. 683-709.

[Sch06] Douglas C SCHMIDT. "Model-driven engineering". In : *COMPUTER-IEEE COMPUTER SOCIETY-* 39.2 (2006), p. 25-31.

[SJ13] Andreas SODERBERG et Rolf JOHANSSON. "Safety contract based design of software components". In : *2013 IEEE international symposium on software reliability engineering workshops (ISSREW)*. IEEE. 2013, p. 365-370.

[SLS18] Eugene SYRIANI, Lechanceux LUHUNU et Houari SAHRAOUI. "Systematic mapping study of template-based code generation". In : *Computer Languages, Systems & Structures* 52 (2018), p. 43-62.

[Spe+10] Thomas SPENKUCH, Stephen TURNOCK, Matteo SCARPONI et Ajit SHENOI. "Real time simulation of tacking yachts : how best to counter the advantage of an upwind yacht". In : *Procedia Engineering* 2.2 (2010), p. 3305-3310.

[Ste+10] Roland STELZER, Karim JAFARMADAR, Hannes HASSLER et Raphael CHARWOT. "A Reactive Approach to Obstacle Avoidance in Autonomous Sailing". In : *3rd International Robotic Sailing Conference (IRSC)*. 2010.

[Szy02] Clemens SZYPERSKI. *Component software : Beyond object-oriented programming*. 2e éd. Addison-Wesley, 2002.

[Tak+16] Kenta TAKAYA, Toshinori ASAI, Valeri KROUMOV et Florentin SMARANDACHE. "Simulation environment for mobile robots testing using ROS and Gazebo". In : *20th International Conference on System Theory, Control and Computing (ICSTCC)*. IEEE. 2016, p. 96-101.

[Tha+17] Jürgen THANHOFER-PILISCH, Alexander LANG, Michael VIERHAUSER et Rick RABISER. "A systematic mapping study on DSL evolution". In : *2017 43rd Euromicro Conference on Software Engineering and Advanced Applications (SEAA)*. IEEE. 2017, p. 149-156.

[Thr10] Kleanthis THRAMBOULIDIS. "The 3+1 SysML view-model in model integrated mechatronics". In : *Journal of Software Engineering and Applications* 3.02 (2010), p. 109.

[Tom10] Mirosław Tomera. "Discrete Kalman filter design for multivariable ship motion control : experimental results with training ship". In : *Joint Proceedings of Gdynia Maritime Academy & Hochschule Bremerhaven* 26 (2010), p. 34.

[Val11] Fernando Valles-Barajas. "A survey of UML applications in mechatronic systems". In : *Innovations in Systems and Software Engineering* 7.1 (2011), p. 43-51.

[Vid+09] Jorgiano Vidal, Florent De Lamotte, Guy Gogniat, Philippe Soulard et Jean-Philippe Diguet. "A co-design approach for embedded system modeling and code generation with UML and MARTE". In : *2009 Design, Automation & Test in Europe Conference & Exhibition*. IEEE. 2009, p. 226-231.

[VKV00] Arie Van Deursen, Paul Klint et Joost Visser. "Domain-specific languages : An annotated bibliography". In : *ACM Sigplan Notices* 35.6 (2000), p. 26-36.

[Vol+01] Richard Volpe, Issa Nesnas, Tara Estlin, Darren Mutz, Richard Petras et Hari Das. "The CLARAty architecture for robotic autonomy". In : *2001 IEEE Aerospace Conference Proceedings (Cat. No. 01TH8542)*. T. 1. IEEE. 2001, p. 1-121.

[Völ+13] Markus Völter, Thomas Stahl, Jorn Bettin, Arno Haase et Simon Helsen. *Model-driven software development : technology, engineering, management*. John Wiley & Sons, 2013.

[WHR14] Jon Whittle, John Hutchinson et Mark Rouncefield. "The state of practice in model-driven engineering". In : *IEEE software* 31.3 (2014), p. 79-85.

[XJ13] Lin Xiao et Jerome Jouffroy. "Modeling and nonlinear heading control of sailing yachts". In : *IEEE Journal of Oceanic engineering* 39.2 (2013), p. 256-268.

UNIVERSITE BRETAGNE LOIRE / MATHSTIC

UBO
Université de Bretagne Occidentale

Titre : AMSA, un framework dédié à la simulation de lois de contrôle pour des voiliers de compétition

Mots clés : simulation, pilote automatique, architecture logicielle, ingénierie dirigée par les modèles

Résumé : Les voiliers de compétition sont des véhicules complexes et instables qui requièrent un contrôle fiable et robuste. Si la simulation semble un outil indispensable pour la mise au point des lois de contrôle, elle entraîne des problématiques de génie logiciel comme l'intégration de composants hétérogènes au sein du simulateur ou encore la facilité d'exploitation de celui-ci.

Ce travail propose un *framework* logiciel dédié à la simulation de lois de contrôle pour les voiliers de compétition. Ce *framework* s'appuie sur des modèles et des outils facilitant la mise en place d'une architecture logicielle de simulateur. Un style architectural, fondé sur des interfaces de communication standardisées et sur des adaptateurs universels, apporte une réponse au problème de gestion de l'hétérogénéité des composants. Un mécanisme de scénarios est proposé pour faciliter la gestion des simulations. Un scénario permet la paramétrisation des différents constituants d'un simulateur et la production de résultats exploitables en faisant intervenir des événements et des observateurs. La mise en place de balayages automatiques de paramètres est également proposée afin de faciliter la mise au point des modèles et des lois de contrôle.

Divers modèles d'environnement, de bateaux et de lois de contrôle ont été intégrés au *framework* sous forme de composants logiciel, ce qui a permis de réaliser plusieurs simulations ayant des objectifs variés : validation de modèle paramétrique de voilier, rejeu de situation réelle, mise au point de lois de contrôle. Les résultats des simulations valident les modèles utilisés et permettent de confirmer des réglages de pilote obtenus empiriquement lors de navigations.

Title: AMSA, a framework dedicated to simulation of control laws for racing sailboats

Keywords: simulation, autopilot, software architecture, model driven engineering

Abstract: Racing sailboats are complex and unstable vehicles, which require reliable and robust control. If simulation seems to be an essential tool to tune control laws, it comes with software engineering issues, like integration of heterogeneous components in the simulator, or ease of simulator's exploitation.

This work proposes a software framework dedicated to simulation of control laws for racing sailboats. The framework provides models and tools to facilitate the setup of the simulator architecture. An architectural style, based on standard communication interfaces and universal adaptors, answers to the components' heterogeneity issue. A mechanism of scenarios is proposed to facilitate the simulations' management. A scenario allows the parametrization of the different simulator constituents and the production of workable results thanks to events and observers. Automatic parameter sweeping is also possible to facilitate the tuning of models and control laws.

Various models of environment, boats and control laws have been integrated to the framework as software components. That allows to run several simulations with various goals: validation of sailboat parametric model, replay of real situation, tuning of control laws. The simulation results validate the models used and confirm some pilot settings obtained empirically during real navigation.