Un framework d'aide au déploiement et à la personnalisation des systèmes temps réel : application aux autopilotes de drones

Soulimane Kamni

THÈSE

pour l'obtention du Grade de
Docteur de l'Ecole Nationale Supérieure de Mécanique et d'Aérotechnique

(Diplôme National – Arrêté du 25 mai 2016 Modifié par l'Arrêté du 26 Août 2022)

École Doctorale :
MIMME Mathématiques, Informatique, Matériaux, Mécanique, Énergétique
Secteur de Recherche : Informatique et applications

Présentée par :

Soulimane KAMNI

Un Framework d'aide au déploiement et à la personnalisation des systèmes temps réel : application aux autopilotes de drones

Directeur de thèse : **Emmanuel GROLLEAU**
Co-encadrant de thèse : **Yassine OUHAMMOU**
Co-encadrant de thèse : **Antoine BERTOUT**

Soutenue le 20 décembre 2023
devant la Commission d'Examen

JURY

Présidente
Annie CHOQUET-GENIET — Professeur, Université de Poitiers, Poitiers

Rapporteurs
Jean-Philippe BABAU — Professeur, Université de Bretagne Occidentale, Brest
Pierre-Emmanuel HLADIK — Maître de Conférences HDR, École Centrale de Nantes, Nantes

Membre du jury
Jérôme ERMONT — Maître de Conférences, INP-ENSEEIHT, Toulouse
Emmanuel GROLLEAU — Professeur, ISAE-ENSMA, Poitiers
Antoine BERTOUT — Maître de Conférences, Université de Poitiers, Poitiers

Résumé

Cette thèse porte sur la conception des systèmes embarqués temps réel critiques. Elle se concentre plus particulièrement sur la phase cruciale de déploiement, où les fonctions du système sont allouées aux tâches logicielles.

La thèse débute par une présentation approfondie du contexte des systèmes embarqués temps réel, de leurs spécificités et de leur cycle de vie logiciel. Puis, le domaine de l'ordonnancement temps réel est introduit, en détaillant les modèles de tâches, les algorithmes d'ordonnancement et les tests de validation temporelle.

Sur ces bases, deux contributions originales sont proposées pour faciliter le déploiement dans un contexte où les informations sur le comportement temporel des fonctions restent limitées. La première est la méthode RYM, une technique de pré-déploiement des systèmes mono-processeur basée sur les rythmes d'activation. Un outil logiciel est également développé pour valider cette approche. La seconde contribution est un framework pour extraire, modéliser et analyser l'architecture interne des autopilotes open-source de drones à partir du code source. Ce framework est appliqué pour adapter une technique avancée d'ordonnancement à l'autopilote Paparazzi.

Ces deux contributions apportent des réponses au problème difficile du déploiement des systèmes embarqués critiques avec des informations partielles. Elles s'appuient toutes les deux sur des développements de prototypes ouverts basés ingénierie dirigée par les modèles, permettant une adaptation rapide à d'autres outils d'analyse.

Mots-clefs : systèmes embarqués temps réel, conception, déploiement, ingénierie dirigée par les modèles, ordonnancement temps réel.

Abstract

This thesis focuses on the design of real-time critical embedded systems. It concentrates more specifically on the crucial deployment phase, where system functions are allocated to software tasks.

The thesis begins with an in-depth presentation of the context of real-time embedded systems, their specificities and their software lifecycle. Then, the field of real-time scheduling is introduced, detailing task models, scheduling algorithms and timing validation tests.

On these foundations, two original contributions are proposed to facilitate deployment in a context where information on the temporal behavior of functions remains limited. The first is the RYM method, a single-processor pre-deployment technique based on activation rhythms. A software tool is also developed to validate this approach. The second contribution is a framework to extract, model and analyze the internal architecture of open-source drone autopilots from the source code. This framework is applied to adapt an advanced scheduling technique to the Paparazzi autopilot.

These two contributions provide answers to the difficult problem of deploying critical embedded systems with partial information. They are both based on open prototyping developments based on model-driven engineering, allowing rapid adaptation to other analysis tools.

Keywords: real-time embedded systems, design, deployment, model-driven engineering, real-time scheduling.

Secteur de recherche : Informatique et applications

LABORATOIRE D'INFORMATIQUE ET D'AUTOMATIQUE POUR LES SYSTÈMES
École Nationale Supérieure de Mécanique et d'Aérotechnique
Téléport 2 – 1 avenue Clément Ader BP 40109 86961 Futuroscope Chasseneuil Cedex
Tél. +33 (0)5 49 49 80 80 Fax : +33 (0)5 49 49 80 00

Mots clés RAMEAU :

Drones / Drone aircraft

Durée de vie (ingénierie) / Service life (Engineering)

Ingénierie dirigée par les modèles / Model-Driven Engineering

Ordonnancement (informatique) / Computer scheduling Systèmes, Conception de / System design Systèmes embarqués (informatique) / Embedded computer systems

Temps réel (informatique) / Real-time data processing

Mots clés libres :

Déploiement de systèmes, Framework, Méthode RYM

Remerciements

Cette thèse représente l'aboutissement de nombreuses années de travail acharné et l'accomplissement d'un rêve qui me tenait à cœur. Je suis extrêmement fier d'être arrivé au bout de ce marathon intellectuel et je mesure la chance que j'ai de pouvoir célébrer cette réussite avec mes proches.

Je tiens à remercier chaleureusement mon directeur de thèse, Emmanuel GROLLEAU, dont la rigueur scientifique et les conseils pointilleux m'ont permis d'élever mon travail. Yassine OUHAMMOU, qui par ses connaissances approfondies du sujet et sa vivacité d'esprit, a su me guider aux moments opportuns. Et Antoine BERTOUT, dont la bienveillance, la disponibilité et le réconfort dans les moments de doute ont été précieux.

Je souhaite adresser mes plus sincères remerciements aux membres du jury, qui ont grandement contribué à l'évaluation rigoureuse de ce travail par leur expertise et leur lecture attentive. Je tiens à exprimer ma gratitude envers professeur Annie CHOQUET-GENIET, présidente du jury et examinatrice, ainsi que professeur Jean-Philippe BABAU et docteur Pierre-Emmanuel HLADIK, rapporteurs de la thèse, pour leur temps consacré à évaluer attentivement mon travail et pour leurs commentaires constructifs. Un grand merci également à docteur Jérôme ERMONT d'avoir accepté de faire partie du jury chargé d'évaluer mes travaux de recherche.

J'adresse mes plus sincères remerciements à mes collègues enseignants du département d'informatique et d'automatique, qui m'ont accueilli chaleureusement parmi eux pendant mes années d'attaché temporaire d'enseignement et de recherche.

Je souhaite adresser mes plus sincères remerciements à Ladjel BELLATRECHE pour son aide précieuse qui a grandement contribué à l'avancée de cette thèse. Son énergie communicative au sein du laboratoire a été une source de motivation constante.

Je tiens à remercier chaleureusement Amin MESMOUDI pour ses encouragements et pour m'avoir soutenu à continuer dans les moments les plus difficiles de cette thèse. Son soutien a été essentiel pour persévérer et mener à bien ce travail.

Je souhaite également adresser ma gratitude à notre dévouée secrétaire Bénédicte BOINOT pour son aide précieuse dans toutes les démarches administratives liées à la préparation et à la soutenance de cette thèse. Sa réactivité et son efficacité m'ont fait gagner un temps considérable.

Je tiens à exprimer ma reconnaissance à Mickael BARON, pour les nombreuses discussions enrichissantes que nous avons eues et ses précieux conseils qui m'ont guidé pendant cette thèse.

Un grand merci à tous mes collègues de recherche avec qui j'ai eu la chance de travailler au cours de cette aventure. Vos discussions, votre camaraderie et votre soutien mutuel ont été une source d'inspiration constante. Ensemble, nous avons surmonté les défis et célébré les réussites, créant ainsi des souvenirs inoubliables.

À mes amis, Abdelkrim BOUDAOUD, Zakaria MADI, Boumedienne SAIDI, Nadir GUERMOUDI, Abderrezzaq SENDJASNI, Stephan DE BRA, Abdellah KHELIL et tous les autres, qui m'ont soutenu avec amour et compréhension tout au long de ce parcours académique exigeant, je vous remercie du fond du cœur. Votre présence, vos encouragements et les moments de détente que nous avons partagés ont été essentiels pour maintenir mon équilibre et ma motivation.

Enfin, je souhaite dédier cette thèse à ma famille. Votre soutien inconditionnel, votre amour et votre foi en moi ont été la force qui m'a incité à persévérer même dans les moments les plus difficiles. Cette réalisation est aussi la vôtre. Je suis profondément reconnaissant envers chacun d'entre vous, et cette thèse porte l'empreinte de vos contributions, de votre soutien. Merci du fond du cœur pour avoir fait partie de ce voyage académique.

Table des matières

Liste des figures

Liste des tableaux

Liste des abréviations

RTES Real-Time Embedded System

ADL Architecture Description Language

AADL Architecture Analysis and Design Language

ARCADIA ARChitecture Analysis and Design Integrated Approach

AUTOSAR AUTomotive Open System ARchitecture

BCET Best-Case Execution Time

CWM Common Warehouse Metamodel

DAG Directed Acyclic Graph

DM Deadline Monotonic

DRT Diagraph Real-time Task

DSML Domain-Specific Modeling Language

EAST-ADL Electronic Architecture and Software Tools - Architecture Description Language

ECU Electronic Control Unit

EDF Earliest Deadline First

GMF Generalized Multiframe

HMPSoC Heterogeneous Multiprocessor System-on-Chip

IDM Ingénierie Dirigée par les Modèles

LLF Least Laxity First

MARTE Modeling and Analysis of Real-time Embedded Systems

MDA Model Driven Architecture

MOF Meta-Object Facility

OCL Object Constraint Language

OMG Object Management Group

OPA Optimal Priority Assignment

PCP Priority Ceiling Protocol

PIM Platform Independent Model

PIP Priority Inheritance Protocol

PSM Platform Specific Model

QVT Query/View/Transformation

RB Recurring Branching

ROS Robot Operating System

RRT Real-time Recurrent Task

RTA Response Time Analysis

RTOS Real-Time Operating System

RTM Rate Monotonic

SA/RT Structured Analysis for Real-Time

SPT UML Profile for Schedulability, Performance and Time

SRP Stack Resource Policy

SysML Systems Modeling Language

UAV Unmanned Aerial Vehicle

Liste des symboles

T_i Période de la tâche i

D_i Échéance relative de la tâche i

r_i Instant d'activation de la tâche i

C_i Temps d'exécution pire cas de la tâche i

U_i Utilisation de la tâche i

L_i Laxité de la tâche i

$\mathbf{RT}_i$ Temps de réponse de la tâche i

$T_{\mathbf{min}}$ Période minimale

$T_{\mathbf{max}}$ Période maximale

$\mathbf{df}_i$ Flot de données i

F_i Fonction i

E_i Événement externe i

$\mathbf{Hp}(i)$ Ensemble des tâches avec priorité plus élevée que i

$w_i(j)$ Calcul itératif du temps de réponse

CHAPITRE 1

Introduction générale

1.1 Contexte et motivations

Dans le domaine des systèms embarqués temps réel (RTES), le processus de mapping des fonctions vers les nœuds d'exécution et de communication est une tâche fondamentale et complexe. Ce placement, souvent réalisé lors de la phase de déploiement, est une étape cruciale pour garantir qu'un système réponde à ses exigences fonctionnelles et extra-fonctionnelles, notamment la latence, les délais de bout en bout, la charge du processeur et le débit du réseau. Cependant, la complexité de cette tâche est exacerbée par le manque de connaissances complètes sur le comportement du système, en particulier dans des cycles de vie tels que AUTOSAR et ARCADIA, où des paramètres clés tels que les durées d'exécution et les délais locaux des tâches restent inconnus à l'étape du mapping. Cette thèse vise à explorer la généralisation du processus de placement des fonctions dans les tâches, tout en examinant comment nos hypothèses concernant nos connaissances et incertitudes s'harmonisent avec les exigences des environnements embarqués critiques.

Dans le paysage contemporain du développement de RTES, les entreprises gardent souvent secrets leurs cycles de développement logiciel et les méthodologies choisies, ne les partageant qu'en interne dans un climat de confidentialité.

Cependant, certaines méthodologies et normes ont acquis une acceptation et une utilisation répandues. Un exemple notable est Capella, un outil basé sur la méthode de conception ARCADIA, qui a suscité une attention considérable depuis sa sortie en 2015. De même, l'industrie automobile s'appuie sur les normes AUTOSAR, qui prescrivent un cycle de vie logiciel où la décomposition fonctionnelle est dérivée des exigences et est ensuite placée (ou *mappée*) vers les tâches. Ces approches ne sont pas uniques ; elles reflètent plutôt une tendance plus large dans l'industrie.

Dans l'approche orientée modèle (MDA), promue par l'Object Management Group (OMG), un modèle métier ou de domaine est initialement présenté dans un modèle indépendant de la plateforme (PIM) avant d'être affiné dans un modèle spécifique à la plateforme (PSM). À ce stade, les concepteurs sont confrontés à une étape cruciale où ils doivent prendre des décisions de conception, souvent avec des informations limitées sur les fonctions elles-mêmes. Ces fonctions font partie de chaînes fonctionnelles entrelacées, des séquences de fonctions interdépendantes qui commencent par des capteurs, des réseaux d'entrée ou des canaux de communication et subissent des transformations avant d'atteindre des actionneurs ou des canaux de sortie. Le contrôle du délai de bout en bout d'une chaîne fonctionnelle devient primordial pour répondre aux exigences de performance non fonctionnelles, nécessitant l'imposition de délais locaux ou de bout en bout sur les fonctions ou les chaînes.

Cette thèse aborde la phase de déploiement, une étape cruciale où la configuration optimale est déterminée en effectuant le mapping des fonctions et des messages vers les nœuds d'exécution et de communication tout en respectant les exigences fonctionnelles et extra-fonctionnelles, notamment la latence, les délais de bout en bout, la charge du processeur et le débit du réseau. Il convient de noter que cette recherche suppose uniquement la connaissance des délais de bout en bout et des rythmes d'entrée possibles des chaînes fonctionnelles, en évitant de s'appuyer sur la durée d'exécution ou les délais locaux des tâches, souvent inconnus dans les cycles de vie AUTOSAR et ARCADIA. La phase de déploiement se présente comme une tâche redoutable et chronophage qui exige une expertise pour parvenir à une solution équilibrée.

1.2 Objectifs de la thèse

L'objectif principal de la thèse est de proposer des solutions pour permettre la validation des systèmes temps réel critiques dès les phases amont du cycle de développement, en particulier pendant la phase de la conception préliminaire. En effet, malgré le manque d'informations à ce stade, il est essentiel de pouvoir vérifier le plus tôt possible les exigences de sûreté de fonctionnement. Pour cela, des solutions basées sur l'ingénierie dirigée par les modèles ont été explorées, cette approche étant adaptée à la conception et au développement de systèmes complexes et critiques devant présenter un haut niveau de fiabilité.

Le second objectif de cette thèse est de fournir un framework basé sur les modèles pour représenter les éléments ontologiques du domaine des drones et plus particulièrement des autopilotes. Ce travail s'inscrit dans le cadre du projet européen Comp4drones [1] et vise à modéliser les connaissances du domaine pour les rendre accessibles et réutilisables.

1.3 Organisation de la thèse

La thèse se compose de six chapitres et elle s'organise comme suit

- **Chapitre 2 : Systèmes embarqués temps réel et conception à base des modèles**

  Ce chapitre présente les concepts fondamentaux liés aux systèmes embarqués temps réel. Il définit ce qu'est un système embarqué temps réel et le positionne par rapport aux systèmes informatiques en général. Puis, il introduit les phases clés de leur cycle de vie en V. Enfin, il présente plusieurs approches méthodologiques pour leur conception, dont AUTOSAR et ARCADIA. L'objectif est de poser les bases sur les systèmes embarqués temps réel qui seront utilisées dans la suite de la thèse.

- **Chapitre 3 : Ordonnancement et validation des systèmes temps réel**

  Ce chapitre introduit le domaine de l'ordonnancement temps réel qui consiste à allouer les ressources aux différentes tâches tout en respectant les contraintes temporelles. Il présente les modèles de tâches, les algorithmes d'ordonnancement sur mono et multiprocesseur, ainsi que les tests d'ordonnançabilité. Ces concepts seront mobilisés dans les chapitres ultérieurs lors de la vérification des solutions de déploiement.

- **Chapitre 4 : RYM : Une approche basée sur les rythmes pour le prédéploiement des RTES critiques**

  Ce chapitre présente notre première contribution avec la méthode RYM, une technique de pré-déploiement des systèmes temps réel critiques. Elle permet de générer une configuration de tâches à partir d'un modèle fonctionnel en se basant sur les rythmes d'activation. Nous détaillons la méthode, réalisons une preuve d'optimalité, et présentons un outil logiciel développé.

[1] https://www.comp4drones.eu/

- **Chapitre 5 : Personnalisation du logiciel d'autopilotes de drone à travers un framework de rétro-ingénierie**

  Ce chapitre présente notre seconde contribution avec un framework pour extraire, modéliser et analyser l'architecture interne des autopilotes open-source de drones. Nous décrivons l'approche pour automatiser l'extraction d'un modèle à partir du code source. Puis, nous montrons comment ce modèle peut être utilisé pour adapter et valider des techniques d'ordonnancement sur un cas concret.

- **Chapitre 6 : Conclusion générale**

  Enfin, ce dernier chapitre résume les contributions apportées dans cette thèse et ouvre des perspectives pour de futurs travaux de recherche.

1.4 Publications scientifiques

Cette thèse a fait l'objet de deux publications scientifiques dans des conférences internationales et une revue internationale. Un quatrième article est en cours de soumission.

1.4.1 Conférences internationales

- Soulimane Kamni, Yassine Ouhammou, Antoine Bertout, Emmanuel Grolleau: Towards a Model-based Multi-Objective Optimization Approach For Safety-Critical Real-Time Systems. DATE 2020 : 634-637 [Kam+20]

- Soulimane Kamni, Yassine Ouhammou, Emmanuel Grolleau, Antoine Bertout, Gautier Hattenberger: A Reverse Design Framework for Modifiable-off-the-Shelf Embedded Systems: Application to Open-Source Autopilots. MEDI 2022 : 133-146 [Kam+22]

1.4.2 Revues internationales

- Soulimane Kamni, Antoine Bertout, Emmanuel Grolleau, Gautier Hattenberger, Yassine Ouhammou: Easing the tuning of drone autopilots through a model-based framework. Journal of Computer Languages (COLA) [Kam+23]

1.4.3 Soumission en cours

- au Journal of Systems and Software (JSS): Soulimane Kamni, Yassine Ouhammou, Emmanuel Grolleau, Antoine Bertout: A rhythm-based pre-deployment method for software synthesis of single-core safety-critical real-time systems.

CHAPITRE 2

Systèmes embarqués temps réel et conception à base des modèles

2.1 Introduction

L'architecture logicielle est un élément clé dans la conception des systèmes embarqués temps réel critiques. Elle doit refléter à la fois les besoins fonctionnels du système et les contraintes liées à la plateforme d'exécution matérielle. De nombreuses approches ont été proposées pour modéliser et concevoir les architectures logicielles, chacune avec ses forces et ses faiblesses.

Dans ce chapitre, nous présentons les principales méthodes et langages utilisés dans l'industrie pour la modélisation architecturale des systèmes embarqués critiques. Nous nous concentrons en particulier sur les normes et langages liés au domaine automobile, comme AUTOSAR et EAST-ADL, ainsi que sur le langage AADL issu du domaine avionique. Un élément clé que partagent ces approches est la séparation entre l'analyse fonctionnelle, qui capture les capacités requises du système, et le placement (ou *mapping*) sur l'architecture matérielle, qui décrit comment les fonctions seront mises en œuvre.

Cette séparation des préoccupations est essentielle pour permettre la réutilisation, la portabilité et l'évolutivité des architectures logicielles. Elle est au cœur des approches dirigées par les modèles comme l'architecture dirigée par les modèles (MDA) et les langages de description d'architecture (ADL).

Nous présentons les concepts clés, les types de composants logiciels et matériels, ainsi que les mécanismes d'interaction et de communication offerts par ces langages. Nous comparons en particulier AADL et EAST-ADL, deux ADLs populaires dans leurs domaines respectifs.

L'objectif de ce chapitre est de montrer comment la modélisation et l'analyse architecturale basée sur des langages formels permettent de répondre aux défis de complexité, de criticité et d'évolutivité des systèmes embarqués temps réel. Les ADLs présentés posent les bases pour l'application de techniques formelles pour la vérification des architectures logicielles.

2.2 Système embarqué temps réel

Un système embarqué temps réel est avant tout un système informatique de contrôle-commande.

2.2.1 Système informatique

Un système informatique est un ensemble de moyens informatiques et de télécommunication ayant pour finalité d'élaborer, traiter, stocker, acheminer, présenter ou détruire des données. Les systèmes informatiques peuvent être classés en trois catégories : les systèmes transformationnels, les systèmes réactifs et les systèmes interactifs.

1. Les systèmes transformationnels transforment les données d'entrée en données de sortie à leur propre rythme, sans interaction avec leur environnement.

Figure 2.1: Illustration d'un système de contrôle temps réel

2. les systèmes interactifs représentent généralement la classe des programmes dotés d'une interface graphique. Ils répondent aux demandes de l'utilisateur, et il est bienvenu qu'ils répondent rapidement.

3. Les systèmes réactifs sont chargés de contrôler les procédés. Un procédé est un système physique à contrôler, évoluant dans son propre environnement. Par conséquent, le système réactif doit réagir assez rapidement par rapport à la dynamique du procédé contrôlé et son environnement. Contrairement aux autres catégories, les systèmes réactifs ne présentent pas de comportements récurrents : le procédé évoluant avec sa propre dynamique dans un environnement souvent complexe, il est rare que les scénarios d'exécution du système réactif/évolution du procédé dans son environnement soient reproductibles.

   Un système embarqué temps réel fait partie de la catégorie des systèmes réactifs.

2.2.2 Système de contrôle-commande

Un système de contrôle-commande est un système informatique de contrôle de procédé qui peut correspondre à n'importe quel système physique contrôlé. Ce dernier est équipé de capteurs et d'actionneurs pour pouvoir le contrôler. Les capteurs (dispositifs d'entrée) sont responsables de la collecte d'informations de l'état du procédé dans son environnement, tandis que les actionneurs permettent au procédé d'agir en fonction de la commande calculée après le traitement des données des capteurs (figure 2.1).

Les dispositifs d'entrée permettent de récupérer des informations sur l'état du procédé à contrôler. Ils peuvent être classés en deux grandes catégories : les dispositifs de type *push* (actifs) et les dispositifs de type *pull* (passifs).

Les dispositifs de type push sont capables de générer des interruptions matérielles lorsqu'un événement se produit, ce qui permet de déclencher suite à la routine de traitement d'interruption la tâche de traitement de cette donnée par le système de contrôle. Cela concerne typiquement les capteurs numériques renvoyant des signaux discrets (fronts montants/descendants, signaux PWM, etc.) ainsi que les données provenant de bus de communication comme Ethernet, CAN, I2C, série, USB, etc.

Les dispositifs de type pull sont passifs, c'est au programme de contrôle d'aller périodiquement lire la valeur mise à disposition sur le port associé au dispositif. C'est le

cas notamment des capteurs analogiques renvoyant une tension ou un courant variant de façon continue avec la grandeur physique mesurée. Le programme doit régulièrement échantillonner cette valeur analogique via un convertisseur analogique-numérique.

Le choix entre dispositifs push et pull dépend principalement de la criticité et de la dynamique de la grandeur physique mesurée. Les interruptions matérielles des dispositifs push permettent une réactivité élevée pour les grandeurs à dynamique rapide ou critiques. La scrutation périodique des dispositifs pull convient pour les grandeurs lentes ou non critiques.

Remarquons que l'on peut sciemment considérer un dispositif push comme un dispositif pull. Dans ce cas, la routine de traitement d'interruption se contente de stocker dans un buffer la valeur reçue du dispositif push. Le buffer sera alors scruté comme si c'était un dispositif pull. Cela peut être utile notamment lorsqu'un dispositif push ne possède pas de durée minimale entre deux événements successifs.

2.2.3 Systèmes embarqués

Un système embarqué est un système informatique dont les moyens de calcul sont embarqués sur le procédé contrôlé. Le fait d'embarquer les moyens de calcul implique, en plus des contraintes d'encombrement (taille, poids, forme), des contraintes de consommation d'énergie et de dissipation de chaleur. De plus, l'alimentation électrique des éléments de calcul est embarquée (batteries, carburant, etc.), et/ou ambiante (panneaux solaires, etc.).

2.2.4 Systèmes temps réel

Système temps réel

Un système temps réel est un système informatique qui doit réagir avec un comportement correct à des événements d'entrée dans des délais.

Les systèmes temps réel sont caractérisés non seulement par leur exactitude fonctionnelle, mais également par l'exactitude temporelle [Dav14]. Par conséquent, un système temps réel doit satisfaire les deux types d'exigences suivants [Sta88]:

- **Exactitude logique :** le résultat généré et calculé par le système doit être correct.
- **Exactitude temporelle :** le comportement correct du système est défini par le respect du délai des sorties.

Dans les applications critiques, un résultat obtenu après le délai est non seulement tardif, mais faux. En fonction des conséquences pouvant survenir du fait d'un délai non respecté, un système temps réel peut être classifié en trois catégories dans le monde académique [But11]:

- **Strict :** un système temps réel est dit "strict" si la production des résultats après leur délai peut avoir des conséquences catastrophiques sur le procédé contrôlé ou son environnement.

- **Ferme :** un système temps réel est dit "ferme" si la production des résultats après leur délai est inutile pour le système, mais ne provoque aucun dommage.
- **Mou :** un système temps réel est dit "mou" si les résultats produits après leur délai ont encore une utilité pour le système, bien que causant une dégradation des performances.

2.3 Cycle de vie de développement des systèmes temps réel

Tout développement logiciel requiert le suivi d'un cycle de vie. Dans le monde critique, le cycle en V (voir figure 2.2) est encore privilégié de nos jours. Ce dernier est également appelé modèle de vérification et de validation, est une méthodologie de développement logiciel qui suit un cheminement séquentiel, tout en autorisant des retours en arrière le cas échéant. Il repose sur la relation entre chaque phase du processus de développement et sa phase de test associée. Le cycle de vie en V met l'accent sur l'importance des tests et de la validation tout au long du processus de développement [Som11].

Le cycle de vie en V est un standard reconnu pour le développement logiciel, normalisé par différentes organisations telles que l'ISO avec les normes ISO/IEC 12207 [ISO08a] et 15288 [ISO08b], l'IEEE avec la norme 12207 [IEE08], le département de la défense américain via la norme MIL-STD-498 [MIL94] ou encore l'EIA et sa norme EIA-632 [EIA03]. Ces standards décrivent les grandes phases du cycle de vie logiciel sous la forme d'un V.

Le cycle de vie en V se décompose plus précisément en plusieurs étapes clés :

1. **Spécification :** il s'agit de définir les exigences du système. La spécification exprime ce que le système doit réaliser et les qualités que le système doit posséder. En entrée de cette phase, on considère des exigences. On y distingue deux catégories : les exigences **fonctionnelles** et **non fonctionnelles** (également appelées exigences extra-fonctionnelles). Tandis que les exigences fonctionnelles définissent les capacités du système, les exigences non fonctionnelles se concentrent sur les performances, la conception et les contraintes de qualité. En particulier, pour les systèmes temps réel, les contraintes temporelles font partie des exigences non fonctionnelles. La spécification peut être exprimée de manière informelle en langage naturel, mais elle peut également être exprimée dans un langage formel ou semi-formel.
2. **Conception :** cette phase cruciale du cycle de développement, il s'agit de répondre aux exigences définies préalablement par la décomposition du système et la description de la structure interne de ce dernier. Elle est réalisée en deux sous-phases complémentaires : la conception **préliminaire** et la conception **détaillée**. La **conception préliminaire** établie une architecture générale du système qui se compose de modules et tâches, sous forme de vues structurelles et comportementales du système. La **conception détaillée** permet de détailler et de raffiner le résultat de la conception préliminaire du système pour obtenir une description détaillée de chaque composant et spécifier les structures de données, les protocoles de communication, le découpage en entités d'exécution parallèle, etc. La conception est réalisée grâce

aux méthodes de conception structurées et adaptées, qui sont souvent basées sur des langages formels, de type DSML.

3. **Implémentation :** le codage réel du système a lieu dans cette phase. Les modules conçus lors de la phase précédente sont implémentés et intégrés pour former le système complet. Les langages autorisant le parallélisme sont souvent utilisés pour implémenter les systèmes temps réel.

4. **Tests unitaires :** permettent de valider chaque composant logiciel, tel que les fonctions et les protocoles. Cette phase vérifie aussi que les vues statique et dynamique détaillées sont cohérentes. Le livrable est la validation de la spécification de conception détaillée.

5. **Tests d'intégration :** valide que l'architecture de haut niveau proposée couvre bien toutes les exigences. Elle vérifie aussi que les composants, modules et tâches sont correctement définis. Cette phase permet également de valider la cohérence entre les vues statique et dynamique. Le livrable est la validation du document de conception préliminaire.

6. **Validation :** les tests de validation permettent de valider le système dans son ensemble par rapport aux exigences utilisateur. Des tests et analyses fonctionnels et non fonctionnels sont effectués. Cette phase valide également le respect des contraintes temps réel avant le déploiement du système.

Le cycle de vie met l'accent sur les tests et la validation précoces pour détecter et corriger les problèmes dès le début du processus de développement. Cependant, il peut être rigide et moins flexible lorsque les exigences changent ou évoluent pendant le cycle de développement.

2.4 Méthodologies et langages de conception standardisés

La conception logicielle est une phase essentielle du processus de développement logiciel qui se concentre sur la création d'un plan détaillé ou d'une feuille de route pour le système logiciel à développer. Elle implique de prendre des décisions concernant l'architecture, les composants, les modules, les interfaces et les structures de données du logiciel.

2.4.1 AUTOSAR

La norme AUTOSAR (AUTomotive Open Systems ARchitecture) a été introduite en 2003 dans le cadre d'un partenariat conjoint entre les équipementiers automobiles et leurs fournisseurs de logiciels et de matériel. Aujourd'hui, AUTOSAR compte plus de 300 partenaires mondiaux [AUT23a] et est donc considéré de facto comme une norme dans le domaine automobile. AUTOSAR définit l'architecture de référence et la méthodologie pour le développement de systèmes logiciels automobiles, et fournit le langage (métamodèle) pour leurs modèles architecturaux.

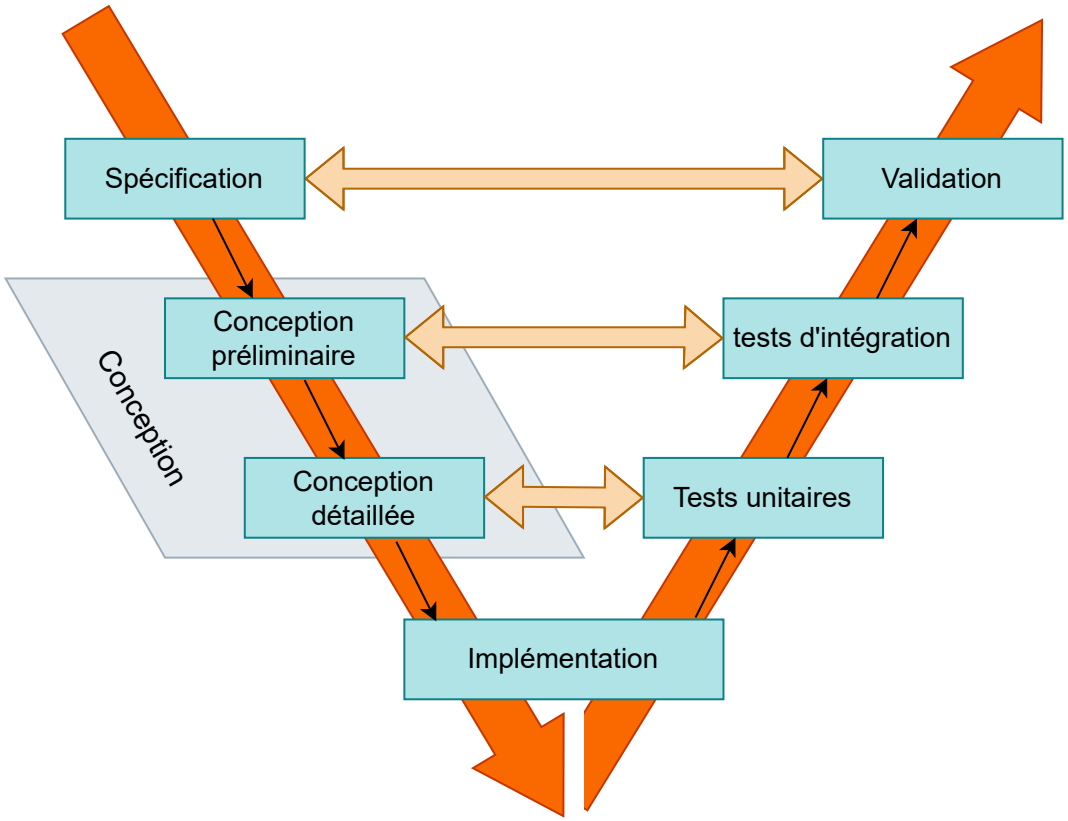

Figure 2.2: Cycle de vie en V

Modèle

Un modèle est une simplification abstraite d'un système, créée dans le but de décrire son fonctionnement de manière à pouvoir répondre aux questions qui lui sont liées. Souvent, un système complexe peut être appréhendé à travers l'utilisation de plusieurs modèles interconnectés.

Métamodèle

Un métamodèle est un modèle qui décrit les éléments et les règles utilisés pour construire des modèles dans un domaine donné.

Comparativement à la norme DO-178C utilisée dans l'avionique civile qui est descriptive, AUTOSAR est une norme prescriptive. En d'autres mots, DO-178C définit un processus, sans donner de règles quant à la mise en œuvre des phases de ce processus. AUTOSAR impose un certain nombre de choix techniques, d'outillages, et de formats d'échange entre les différentes phases.

Concept de base d'AUTOSAR

Le standard AUTOSAR repose sur un ensemble de concepts principaux. Plus précisément, il propose une première vision logicielle abstraite concernant le déploiement sur le matériel. Parmi ces concepts clés, on trouve :

- **Composant logiciel (Software Component - SW-C) :** est un bloc de construction réutilisable du logiciel AUTOSAR. Un composant logiciel AUTOSAR encapsule un ou plusieurs runnables et communique avec son environnement à travers des ports virtuels.
- **Runnable :** est le composant fonctionnel atomique qui ne peut pas être divisé davantage. Il est composé de fragments de codes fonctionnels et peut communiquer avec d'autres runnables dans le même SW-C par variables inter-runnables ou runnables dans les autres SW-C par interface et ports.
- **Bus Fonctionnel Virtuel (Virtual Functional Bus - VFB) :** ce concept représente le moyen de communication des composants logiciels. Un VFB permet la description abstraite des communications et le déploiement de manière indépendante. L'implémentation du VFB peut être réalisée ultérieurement par un support de communication de l'infrastructure matérielle.
- **Unité de Contrôle Électronique (Electronic Control Unit - ECU) :** est généralement une unité électronique basée sur un microcontrôleur qui exécute des logiciels pour gérer différentes fonctions automobiles, telles que le contrôle du moteur, le contrôle de la transmission, le contrôle du châssis, le contrôle de la carrosserie, etc. Chaque ECU fonctionne comme un module séparé responsable d'un ensemble spécifique de tâches.

Architecture

L'architecture AUTOSAR a pour objectif de réduire les problèmes existants au moment de la conception, en facilitant les interactions entre les parties prenantes. Cette architecture est composée de trois couches, à savoir la couche application, la couche logiciel de base et la couche environnement d'exécution, illustrées dans la figure 2.3. Les couches permettent de bien différencier la partie fonctionnelle de la partie matérielle.

- **La couche application :** c'est une couche indépendante du matériel. Elle est composée de SW-C.
- **La couche logiciel de base :** elle représente le logiciel bas niveau de l'architecture. Elle fournit des services aux composants logiciels AUTOSAR de la couche application. Elle est nécessaire pour exécuter la partie fonctionnelle, qui inclut l'un des deux systèmes d'exploitation AUTOSAR et des services tels que les modules de communication, la mémoire, etc. Elle est composée d'une partie dite abstraction du microcontrôleur qui dépend du matériel.
- **La couche environnement d'exécution RTE (Run-time Environnement)** : Cette couche est généralement générée automatiquement, en se basant sur les interfaces des SW-C. Les SW-C sont exécutés sur un microcontrôleur par le biais du RTE grâce aux interfaces de ce dernier. Le RTE veille à ce que les runnables soient invoqués au bon moment, garantissant ainsi une exécution coordonnée et synchronisée.

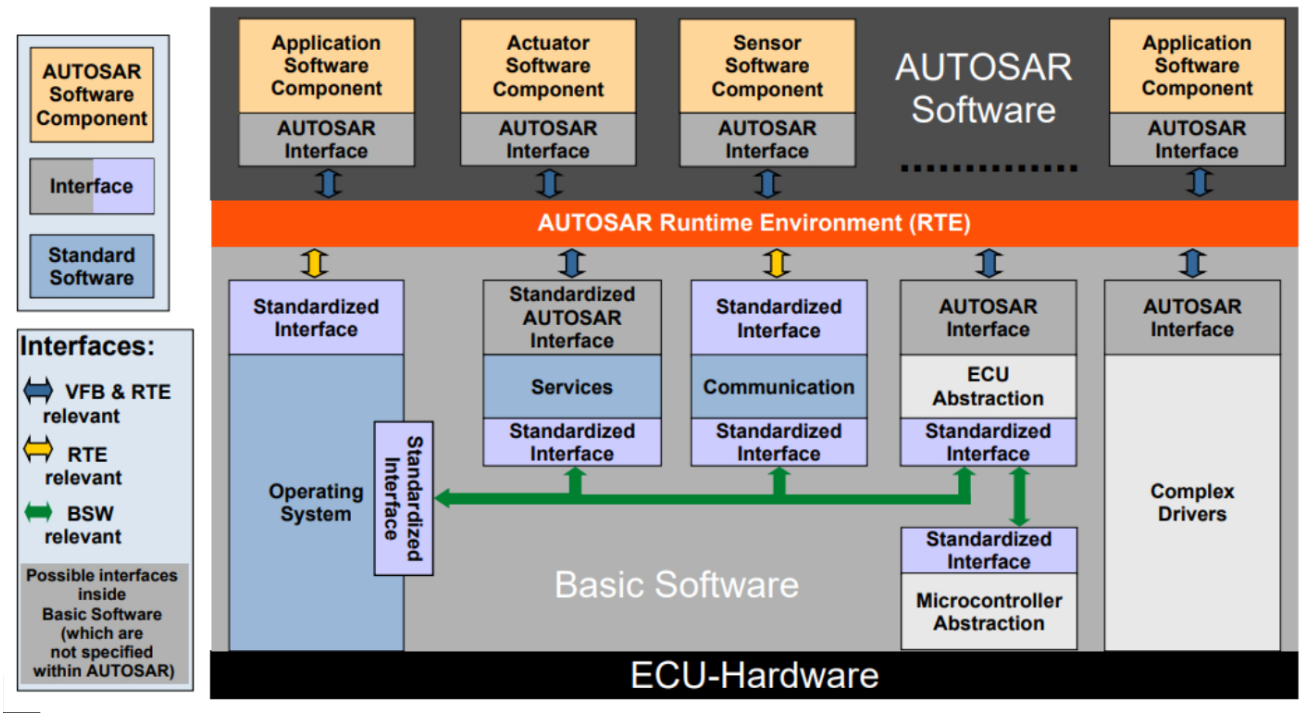

Figure 2.3: L'architecture logicielle AUTOSAR [AUT23b]

Méthodologie

AUTOSAR propose une méthodologie pour la conception et le développement d'un système automobile. Cette approche est basée sur les composants. Elle se compose de 3 étapes comme le montre la figure 2.4:

- **La définition de l'architecture logicielle :** cette étape consiste à décomposer l'application en composants logiciels, décrire leurs comportements internes et définir les interactions entre ces composants grâce aux bus fonctionnels virtuels. En sortie, une description pour chaque SW-Cs est établie.
- **La description des ECUs et la définition des contraintes système :** durant cette étape, les descriptions des ECUs sont établies indépendamment des descriptions des SW-Cs, ainsi que les contraintes systèmes.
- **Le mapping des SW-Cs aux ECUs :** cette étape consiste à affecter les SW-Cs aux ECUs à l'aide des outils permettant le mapping AUTOSAR. L'outil utilisé prend en entrée les descriptions réalisées durant les deux premières étapes. Durant cette étape, les modules RTE et BSW sont générés et configurés au niveau des ECUs.

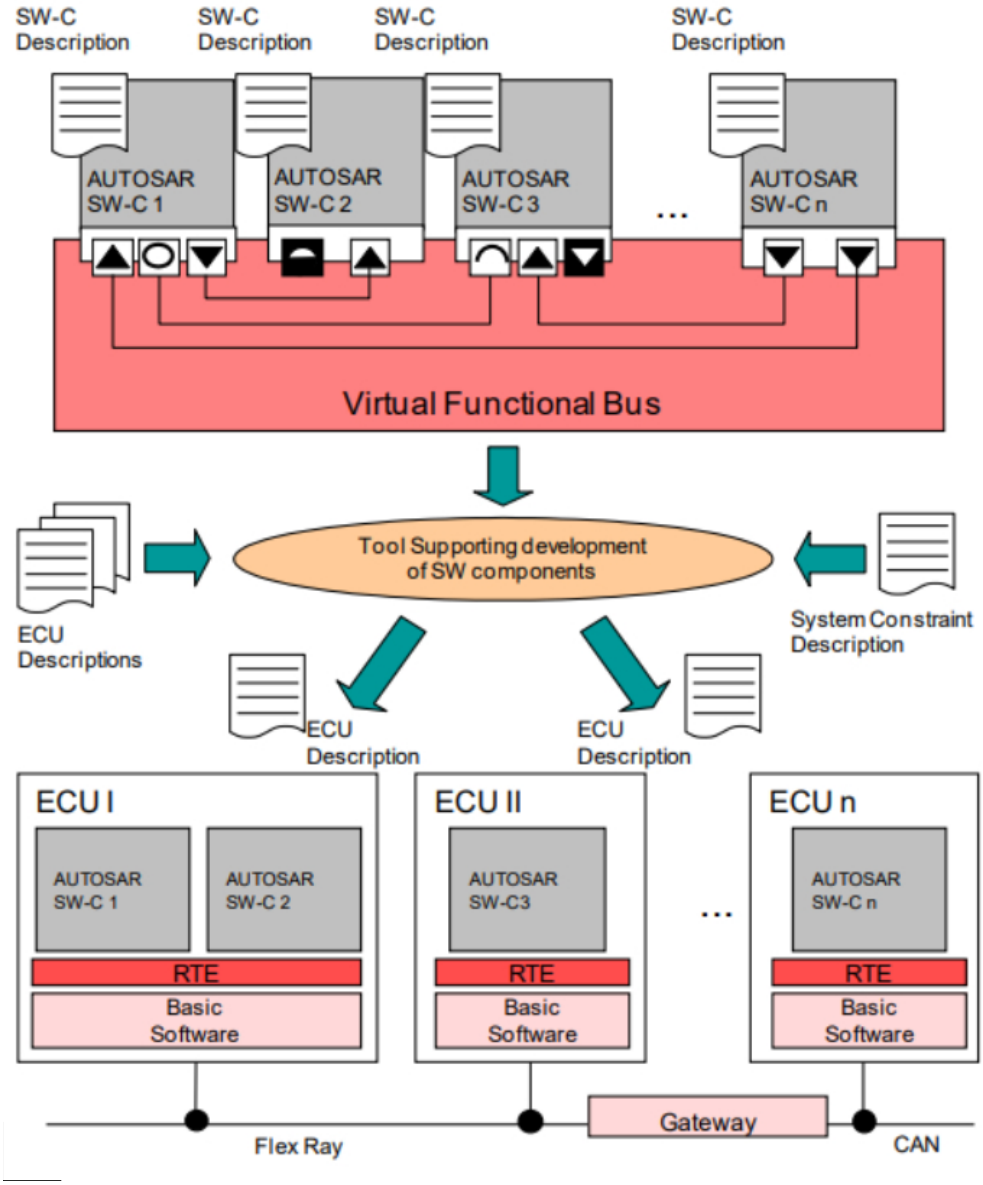

Figure 2.4: Aperçu de l'approche AUTOSAR [AUT23b]

2.4.2 ARCADIA

ARCADIA (ARChitecture Analysis and Design Integrated Approach) est une méthode d'ingénierie basée sur les modèles pour la conception architecturale des systèmes, matériels et logiciels. Elle a été développée par Thales [Gro23b] entre 2005 et 2010 au travers d'un processus itératif impliquant des architectes opérationnels de toutes les activités de Thales (transport, avionique, espace, radar, etc.) [NE04].

Elle assure la mise en œuvre d'une approche structurée sur des phases d'ingénierie successives qui établit une séparation claire entre les exigences (analyse des besoins opérationnels et analyse des besoins du système) bien que celles-ci ne soient pas explicitement prises en compte dans l'outillage présent, et les solutions (architectures logiques et physiques), conformément à la norme IEEE 1220. ARCADIA recommande trois activités interdépendantes obligatoires au même niveau d'importance : Analyse des besoins

et modélisation, Construction des architectures et validation, et Ingénierie des exigences [Roq16].

ARCADIA **DSML** (Domain-Specific Modeling Language) est basé sur les standards UML/SysML [Hau+06] et NAF [nat23] et partage de nombreux concepts avec ces langages. Il facilite l'appropriation par toutes les parties prenantes. ARCADIA est principalement basé sur l'analyse fonctionnelle, puis l'allocation des fonctions aux composants. La richesse d'ARCADIA DSML est comparable à SysML avec une dizaine de types de diagrammes : diagrammes de flux de données, diagrammes de scénarios, diagrammes d'états et de modes, diagrammes de répartition des composants, diagrammes de répartition fonctionnelle, etc.

Langage de modélisation dédié (DSML)

Un *langage de modélisation dédié* (Domain Specific Modeling Language) est un langage de modélisation spécialisé conçu pour être utilisé dans un domaine ou une application spécifique. Les DSML sont utilisés pour créer des modèles spécifiques au domaine, qui sont des représentations de concepts, de processus ou de systèmes spécifiques à un domaine particulier. Les DSML sont souvent utilisés dans le développement de logiciels. Ils peuvent également être employés dans d'autres domaines tels que l'ingénierie, les affaires ou les finances.
Un DSML est structuré en deux concepts, la syntaxe abstraite et la syntaxe concrète :

- la syntaxe abstraite est le métamodèle qui définit l'ensemble des concepts et les relations qu'elles les relient.
- la syntaxe concrète est la représentation graphique ou textuelle des concepts préalablement définis au niveau de la syntaxe abstraite.

Un DSML peut posséder plusieurs syntaxes concrètes pour la même syntaxe abstraite.

AADL (Architecture and Analaysis Design Language) est un DSML de type ADL (Architecture Design Language). AADL est un exemple de DSML destiné à représenter l'architecture logicielle et matérielle d'un système informatique. Il est conçu pour être utilisé par des architectes logiciels/matériels (voir sous-section 2.4.4).

ARCADIA offre une approche structurée pour la conception et le développement des systèmes en mettant l'accent sur l'utilisation de modèles et de méthodes semi-formelles et formelles. ARCADIA favorise la séparation des préoccupations et la décomposition modulaire des systèmes, ce qui permet une meilleure compréhension, analyse et vérification des systèmes.

Les principaux caractéristiques et concepts d'ARCADIA comprennent :

- le développement basé sur les modèles : ARCADIA adopte une approche basée sur les modèles où les aspects du système, les exigences, les fonctions, les comportements et les architectures sont représentés à l'aide de modèles graphiques. Ces modèles

capturent la structure et le comportement du système, facilitant ainsi l'analyse et les décisions de conception.

- la co-ingénierie système et logiciel : ARCADIA reconnaît l'importance de l'interaction entre l'ingénierie système et logicielle. Elle fournit des moyens d'intégrer les modèles système et logiciel, permettant un développement cohérent et coordonné.
- l'analyse fonctionnelle et opérationnelle : ARCADIA met l'accent sur l'importance de l'analyse fonctionnelle et opérationnelle au début du processus de développement. Cela permet d'identifier et de préciser les exigences du système, les interfaces et les scénarios opérationnels.
- les points de vue architecturaux : ARCADIA encourage l'utilisation de points de vue architecturaux pour aborder différentes préoccupations dans la conception du système. Elle offre une approche structurée pour définir et analyser les architectures système, y compris les aspects fonctionnels, physiques et de déploiement.
- la transformation de modèles et génération de code : ARCADIA prend en charge les transformations de modèles et la génération de code pour combler l'écart entre la conception basée sur les modèles et la mise en œuvre. Cela permet la génération automatique de code logiciel à partir des modèles système, garantissant ainsi la cohérence et réduisant les efforts manuels.

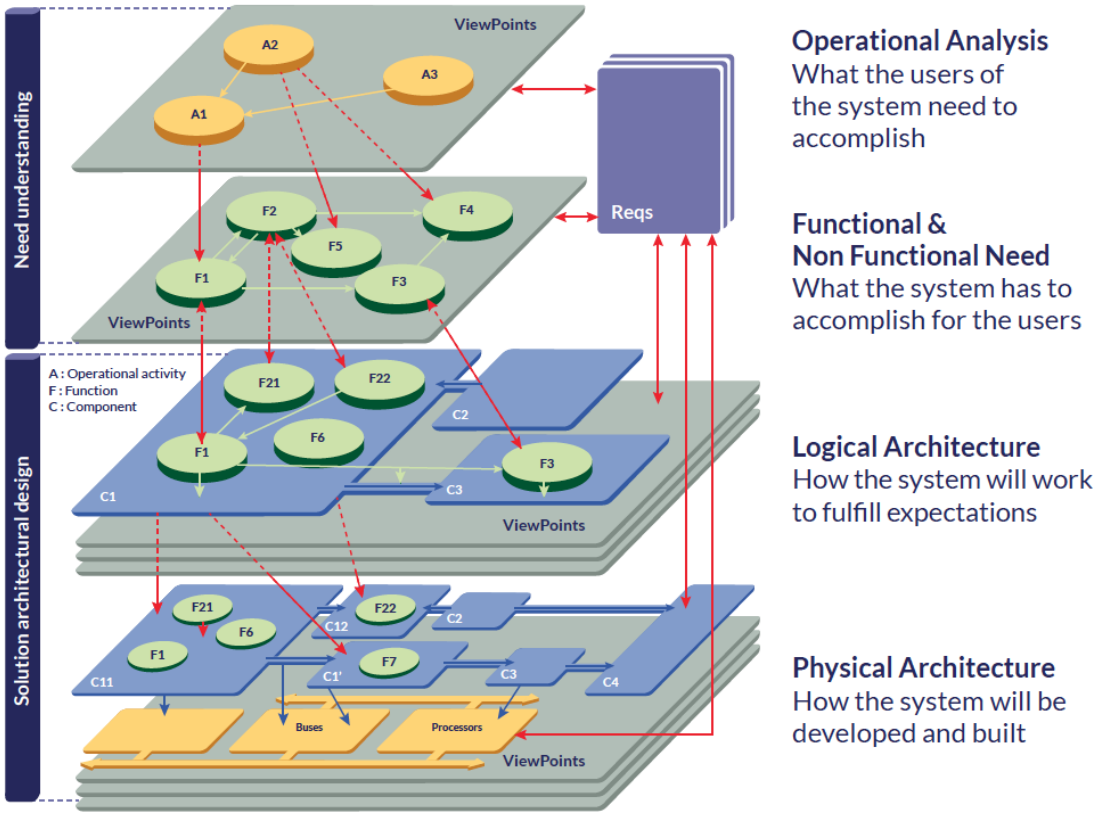

Figure 2.5: Les phases de l'ingénierie d'ARCADIA [Tha23]

ARCADIA se compose de quatre phases, comme le montre la figure 2.5 :

La figure 2.5 illustre les différentes phases du flot de travail ARCADIA qui sont implémentées dans l'outil Capella comme un ensemble d'activité. qui sont :

- **Analyse des besoins opérationnels du client** : définit les besoins des utilisateurs du système à accomplir.
- **Analyse des exigences du système** : définit ce que le système doit accomplir pour ses utilisateurs.
- **Architecture logique** : définit comment le système fonctionnera pour satisfaire les exigences du système.
- **Architecture physique** : définit comment le système sera construit.

ARCADIA n'est pas liée à une organisation de normalisation spécifique. Cependant, elle est conforme aux normes industrielles existantes telles que l'ISO/CEI 42010 pour la description architecturale et les langages de modélisation UML et SysML.

Il est important de noter que bien qu'ARCADIA offre un cadre complet pour l'ingénierie basée sur les modèles, son adoption et son utilisation peuvent varier en fonction de l'industrie spécifique, de l'organisation et des exigences du projet.

Capella

Capella [Pol23] est une application Eclipse [Ecl23] open-source hébergée dans le projet Polarsys. Capella met en œuvre la méthode ARCADIA en fournissant à la fois un DSML (Domain Specific Modeling Language) et un ensemble d'outils dédiés. Il offre un point d'entrée comme un guide méthodologique de la méthode ARCADIA. Cette solution est principalement utilisée pour la modélisation du développement de systèmes complexes et critiques en termes de sûreté, tels que l'aérospatiale, l'avionique, le transport, l'espace, les communications, la sécurité et l'automobile.

2.4.3 Architecture dirigée par les modèles (MDA)

L'Architecture Dirigée par les Modèles (MDA) est une approche de spécification de système et d'interopérabilité basée sur l'utilisation de modèles formels [Obj23; OMG01; DSo01]. L'idée principale est de séparer la spécification du système de son implémentation en utilisant des modèles abstraits indépendants de toute plateforme qui peuvent être mis en correspondance avec des implémentations concrètes dépendant d'une plateforme spécifique.

La MDA a été introduite par l'Object Management Group (OMG) en 2001 en tant que successeur des technologies CORBA précédentes pour l'interopérabilité. Elle vise à placer les modèles système au cœur de l'ingénierie logicielle au lieu des seuls composants et interfaces.

Les normes qui permettent MDA, dont UML, MOF, XMI et CWM, s'appuient sur des spécifications OMG précédentes, mais élèvent les concepts de modélisation comme élément central de l'interopérabilité [OMG01].

Approche MDA

L'approche MDA consiste d'abord à exprimer le comportement et les fonctionnalités du système dans un langage de modélisation indépendant de toute plateforme, comme UML. Ce modèle indépendant de la plateforme (PIM) se concentre purement sur la logique métier et les concepts du domaine. Le PIM est ensuite traduit en un ou plusieurs modèles spécifiques à une plateforme (PSM) par des transformations de modèles. Les PSM ajoutent des détails de plateforme et sont mis en correspondance avec des langages d'implémentation comme Java, .NET, etc.

Cette séparation des préoccupations permet la portabilité, l'interopérabilité et l'abstraction. Le même PIM peut être implémenté sur plusieurs plateformes en définissant des mises en correspondance de plateformes. L'utilisation de modèles formels améliore la précision et l'automatisation.

La MDA repose sur une architecture à 4 niveaux, illustrés par la figure 2.6:

- Niveau 3 (méta-métamodèle) : il s'agit d'un langage utilisé pour créer des méta-modèles faisant partie du niveau 2 du MDA. Le Meta Object Facility (MOF) est le langage fourni par l'OMG pour créer ces métamodèles. Étant considéré comme le dernier niveau du MDA, le méta-méta modèle créé avec ce langage est en conformité avec lui-même, c'est-à-dire qu'il peut s'autodécrire.

- Niveau 2 (méta modèle) : ce sont les métamodèles décrits au niveau 3 et qui définissent les éléments du niveau 1. L'exemple typique pour ce niveau est le méta modèle UML qui décrit la structure interne des modèles UML tels que les classes, les attributs ou les associations.

- Niveau 1 (modèles) : à ce stade, les éléments du monde réel sont décrits, c.-à-d. les éléments du niveau 0, il s'agit, par exemple, d'un diagramme de classe UML dans lequel on spécifie les concepts tels que Client, Véhicule, ainsi que leurs attributs nom, prénom, marque, année, immatriculation, etc.

- Niveau 0 (monde réel) : il s'agit des éléments réels décrits par les modèles du niveau 1. Par exemple, une application codée en JAVA qui contient comme instance de la classe 'Client', appelée 'FOO BAR', un client qui veut acheter un véhicule de marque 'Tesla 2023'.

Outils et langages MDA

Les outils et langages clés utilisés dans MDA sont :

- UML (Unified Modeling Language) est un langage de modélisation et un langage standard de modélisation pour l'ingénierie logicielle orientée objet qui fournit un ensemble de notations graphiques et de sémantique pour exprimer les conceptions de logiciels. UML a été développé par le groupe Object Management Group (OMG) au milieu des années 1990 et est depuis devenu un standard largement utilisé pour la conception et le développement de logiciels. Il comprend une variété de diagrammes, tels que des diagrammes de classes, des diagrammes de cas d'utilisation, des diagrammes de séquence et des diagrammes d'activité, qui sont utilisés pour représenter différents aspects d'un système logiciel.

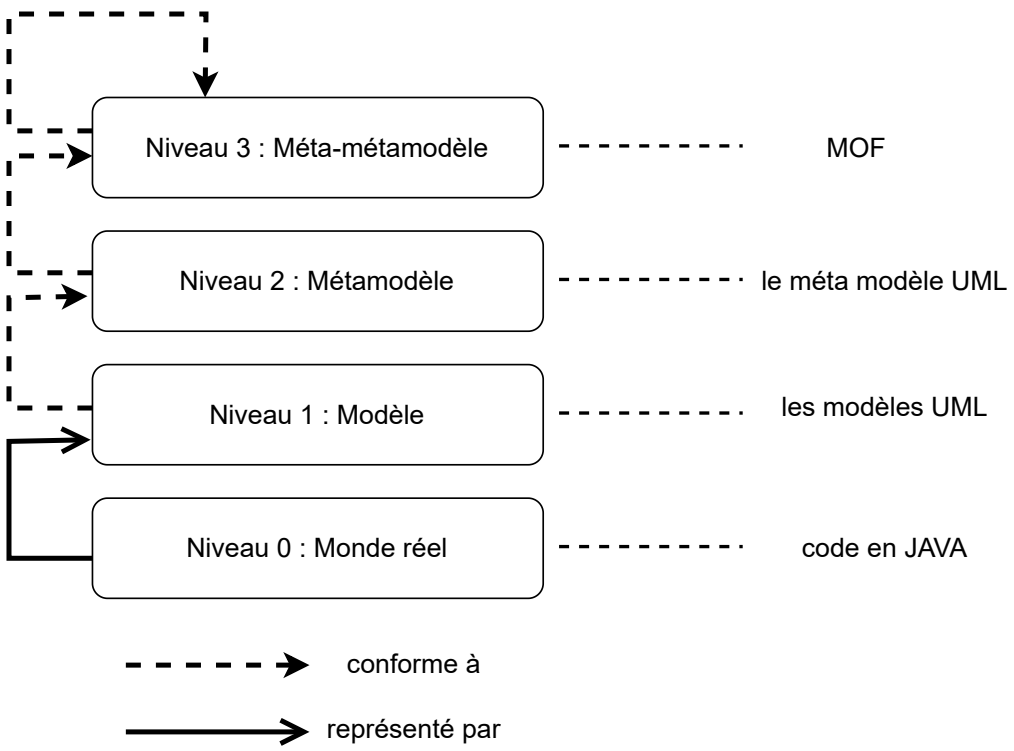

Figure 2.6: Niveaux d'abstraction de la MDA

- MOF (Meta Object Facility) est un langage abstrait, extensible et flexible qui permet de définir et de manipuler des métamodèles. Ce langage est **conforme** à lui-même.

  Notion de conformité

  En ingénierie dirigée par les modèles, la *conformité* fait référence à la relation entre un modèle et son métamodèle, où le modèle adhère précisément à la structure, à la sémantique et aux règles définies dans le métamodèle [KWB03]. Plus précisément, un modèle est conforme à un métamodèle si chaque élément du modèle est une instance d'un élément du métamodèle, et si le modèle respecte toutes les contraintes et règles spécifiées dans le métamodèle [Sei03].

  Le métamodèle définit le langage, les blocs de construction, les relations et les règles pour créer des modèles valides dans un domaine, fournissant la sémantique nécessaire pour des modèles significatifs [AK03].

  Le modèle conforme est une abstraction d'un système construit en utilisant les éléments et les directives établis dans le métamodèle pour représenter une instance valide spécifique [BG01].

- XMI (XML Metadata Interchange) est un standard utilisé pour échanger des métadonnées entre des outils logiciels. Il a été développé pour une utilisation en génie logiciel, en particulier dans le domaine de la modélisation. La normalisation fournie par XMI aide à garantir que les informations échangées entre différents outils sont cohérentes et interopérables, ce qui conduit à une plus grande efficacité et précision dans le processus de développement logiciel.

- CWM (Common Warehouse Metamodel) avec des concepts de domaine réutilisables.
- QVT (Query/View/Transformation) Pour définir des transformations de modèles formelles.

Transformation de modèles

La transformation de modèles est une technique qui consiste à convertir un modèle d'une forme à une autre. La transformation de modèles est un élément clé de la MDA. En effet, elle permet la génération automatique de code ou d'autres artefacts à partir d'un modèle ou l'intégration de modèles provenant de différentes sources ou outils.
Il existe deux types de transformation de modèles : la transformation Model-to-Model (M2M) et la transformation de Model-to-Text (M2T).

Transformations Model-to-Model La transformation M2M nécessite trois éléments en entrée : un métamodèle source *srcMM*, un métamodèle de destination *destMM* et un modèle conforme au métamodèle source. Le modèle est transformé en suivant les règles de la transformation pour obtenir un modèle en sortie qui est conforme au métamodèle de destination.
Les transformations de M2M peuvent être endogènes ou exogènes.

- **Transformations endogènes** : dans le cas d'une transformation endogène, le métamodèle source et le métamodèle de destination sont identiques (c.-à-d. $srcMM = destMM$).

  Un exemple de transformation M2M endogène serait la transformation d'un métamodèle pour les machines à états en un métamodèle pour les réseaux de Petri (considérés comme modèles dans cet exemple), tous deux exprimés en utilisant l'outil ECore. Dans ce cas, les deux modèles sont conformes au métamodèle ECore.

- **Transformations exogènes** : dans le cas d'une transformation exogène, le métamodèle source et le métamodèle de destination sont différents (c.-à-d. $srcMM \neq destMM$).

  Par exemple, la transformation d'un modèle source qui modélise une alarme selon le métamodèle de machine à états en un modèle de destination qui est conforme au métamodèle de réseau de Pétri.

Transformations Model-to-Text il s'agit de produire un ou plusieurs fichiers contenant du texte à partir du modèle donné en entrée. Ce type de transformation est souvent utilisé pour générer du code source ou la documentation d'un modèle.

- OCL - Object Constraint Language pour des modèles précis

Ces éléments fournissent une base solide pour une adoption efficace des principes MDA. Le mapping de modèles avec des plateformes d'implémentation comme Java et .NET a également beaucoup progressé.

Discussion

Dans la conception des systèmes complexes, la séparation entre l'analyse fonctionnelle et le mapping sur l'architecture matérielle est un principe clé. Elle permet de distinguer ce que le système doit accomplir de la façon dont ses fonctions seront mises en œuvre sur des composants physiques. Cette séparation des préoccupations offre de nombreux avantages en termes de flexibilité, portabilité et interopérabilité. Les approches méthodologiques en ingénierie système promeuvent cette distinction entre le modèle fonctionnel et l'architecture matérielle.

L'architecture dirigée par les modèles (MDA) définit une frontière claire entre les modèles indépendants de la plateforme (PIM) qui spécifient la fonctionnalité du système, et les modèles spécifiques à la plateforme (PSM) qui mappent la fonctionnalité aux composants matériels [Gro14]. Le PIM se concentre purement sur les capacités opérationnelles, dissociées des détails de mise en œuvre. Cette séparation permet la portabilité, car le PIM peut être transformé en PSM ciblant différentes plateformes.

De même, AUTOSAR sépare l'architecture logicielle de l'architecture matérielle [AUT22]. L'architecture logicielle comprend des composants fonctionnels, leurs interfaces et leurs interactions. Cela est spécifié indépendamment de la topologie du réseau d'ECU matériel. Des interfaces normalisées entre les composants logiciels permettent une flexibilité dans le mapping du logiciel vers les ECU.

La méthode ARCADIA distingue également le monde fonctionnel du monde physique [RV16]. L'analyse fonctionnelle identifie les fonctions logiques, les flux de données et le séquençage. Cela reste séparé des détails d'implémentation physique. Les architectes peuvent optimiser et analyser la conception fonctionnelle sans contraintes matérielles.

Dans les trois approches, l'analyse fonctionnelle décrit ce que le système doit faire, sans tenir compte de la façon dont les fonctions s'exécutent sur le matériel. Cette séparation des préoccupations est un élément clé pour l'évolutivité, la flexibilité et la portabilité dans les systèmes complexes. Le modèle fonctionnel indépendant du matériel fournit une abstraction stable qui peut être mappée sur diverses plateformes.

2.4.4 AADL

AADL (Architecture Analysis and Design Language) est un langage de modélisation pour spécifier et analyser les architectures logicielles et système [Fei+04]. Il fournit des abstractions à la fois pour les composants logiciels comme les threads (tâches), processus, composants de données, et sous-programmes, ainsi que pour les composants de plateforme d'exécution comme les processeurs, bus, mémoire, et périphériques. AADL supporte l'analyse des propriétés critiques de performance, incluant le timing, l'ordonnancement, la fiabilité, et plus. Il permet l'analyse précoce d'architectures partielles pendant la conception.

Un modèle AADL consiste en des déclarations de type de composant qui spécifient les interfaces, et des déclarations d'implémentation de composant qui spécifient la structure interne (sous-composants), les connexions entre sous-composants, les flux, les modes, les propriétés. Les modèles peuvent être représentés textuellement dans une syntaxe précisément définie, et graphiquement en utilisant des symboles distincts pour chaque type de composant. Les concepts clés dans AADL incluent les composants, caractéristiques,

connexions, flux, modes, propriétés et packages. AADL peut être étendu à travers des propriétés définies par l'utilisateur, des annexes pour définir de nouveaux langages de modélisation, et des patrons de conception.

Dans l'ensemble, AADL est bien adapté pour le développement et l'analyse basés sur le modèle des systèmes embarqués complexes et critiques au niveau des performances [Fei+04].

Historique et standardisation

Les racines d'AADL remontent à la fin des années 1990, lorsque des projets de recherche ont été lancés au Software Engineering Institute (SEI) et l'Agence Spatiale Européenne (ESA) pour définir un langage de modélisation d'architecture pour les systèmes aérospatiaux et avioniques. AADL trouve ses racines dans des langages de description d'architecture (ADL) antérieurs comme MetaH développé par Honeywell. Après les langages prototypes initiaux, l'effort s'est consolidé dans le développement de la norme AADL [SAE06].

En novembre 2004, la Society of Automotive Engineers (SAE) a publié AADL en tant que Norme Internationale SAE AS5506 [SAEb]. AADL a été initialement publié avec les concepts de modélisation de base. Des normes additionnelles ont depuis été publiées pour étendre AADL, incluant l'Error Model Annex pour l'analyse de fiabilité. Le comité de standardisation AADL continue de peaufiner le langage et d'introduire de nouvelles capacités.

AADL a connu une adoption croissante dans l'aérospatial, l'automobile, le ferroviaire, les systèmes médicaux, et d'autres domaines embarqués. Des outils de modélisation open source et commerciaux supportent AADL, et c'est le focus de multiples projets de recherche internationaux. Une conférence internationale annuelle est dédiée à AADL.

Aperçu du langage

AADL permet de modéliser les architectures logicielles et matérielles. Les composants clés incluent threads, processus, processeurs, bus, périphériques, etc. Il supporte la modélisation des aspects temporels, de sûreté, de tolérance aux pannes, de sécurité, etc. Il possède une syntaxe concrète textuelle et une graphique, bien que la syntaxe graphique ne permette pas d'exprimer tous les concepts offerts. Le langage est extensible via des propriétés et des annexes.

Outils et adoption

Un ensemble d'outils open source, OSATE [OSA23], est disponible ainsi que des outils commerciaux comme STOOD, TASTE etc. AADL a été utilisé sur des systèmes avioniques par Airbus, Honeywell, Rockwell Collins etc. Il commence également à être adopté dans le domaine de l'automobile où il est à la base d'EAST-ADL, ainsi que dans le domaine industriel et le domaine médical notamment. Le métamodèle standardisé permet l'interopérabilité entre outils.

Avantages

AADL offre plusieurs avantages clés pour le développement de systèmes embarqués basé sur des modèles :

- **Modélisation formelle et analyse des architectures système** : les sémantiques précisément définies d'AADL permettent la modélisation formelle des architectures système. Cela permet une analyse rigoureuse des propriétés critiques du système à l'aide de la simulation, de la vérification de modèle, de l'analyse d'odonançabilité, etc. Les erreurs peuvent être détectées tôt avant la mise en œuvre.
- **Simulation des aspects temporels et de sécurité dès la conception précoce** : le support d'AADL pour la modélisation des aspects temporels permet la simulation de la planification, des latences de bout en bout, etc. dès les premières étapes de la conception. La modélisation de la sécurité permet l'analyse des dangers, des modes de défaillance et de leurs effets. Cela permet une ingénierie prédictive du système.
- **Évolution des modèles tout au long du cycle de développement** : AADL prend en charge le raffinement incrémental des modèles, des spécifications abstraites aux implémentations détaillées. Cela permet de gérer l'évolution du modèle tout au long du cycle de développement à mesure que la conception mûrit.
- **Génération de code et intégration automatisées** : les sémantiques d'exécution d'AADL permettent la génération automatique de code d'interface et d'intégration système à partir des modèles. Cela réduit les efforts de codage manuel et garantit la cohérence entre la conception et la mise en œuvre.
- **Échange standardisé de modèles entre les outils** : le métamodèle normalisé d'AADL permet l'échange de modèles entre différents outils de modélisation, de simulation, d'analyse et de génération de code. Cela améliore l'interopérabilité des outils.

Représentations Graphiques

AADL définit une notation graphique standard pour représenter visuellement les modèles. Chaque type de composant AADL a un symbole graphique associé. Cette notation permet de représenter clairement la hiérarchie structurelle et les connexions entre les composants.

Les représentations graphiques sont utilisées conjointement avec la forme textuelle. Les outils AADL supportent en général les deux formes et permettent de passer facilement entre elles. La figure 2.7 montre les symboles graphiques pour les composants d'AADL.

Types de composants AADL Le langage d'analyse et de conception architecturale (AADL) définit plusieurs types de composants intégrés, incluant processus, thread, groupe de threads, données, sous-programme, processeur, mémoire, bus, périphérique et système. Par exemple, un composant thread représente une unité ordonnançable d'exécution concurrente, tandis qu'un composant de données représente un stockage de données accessible

par d'autres composants. AADL permet également la modélisation de familles de composants via un mécanisme d'extension, où des composants personnalisés peuvent hériter des caractéristiques de composants standards.

Interfaces de composants AADL Le standard AADL définit plusieurs types d'interfaces que les composants peuvent utiliser pour interagir avec d'autres composants. Il s'agit notamment des interfaces de ports pour le transfert de données et d'événements, des interfaces d'accès pour se connecter à des composants de données, des interfaces d'accès bus pour connecter des composants matériels, des interfaces de sous-programmes pour invoquer des opérations sur des composants, et des interfaces de paramètres pour passer des données aux sous-programmes. Par exemple, un composant thread peut avoir des interfaces de ports d'événements et de données pour recevoir des entrées et envoyer des sorties.

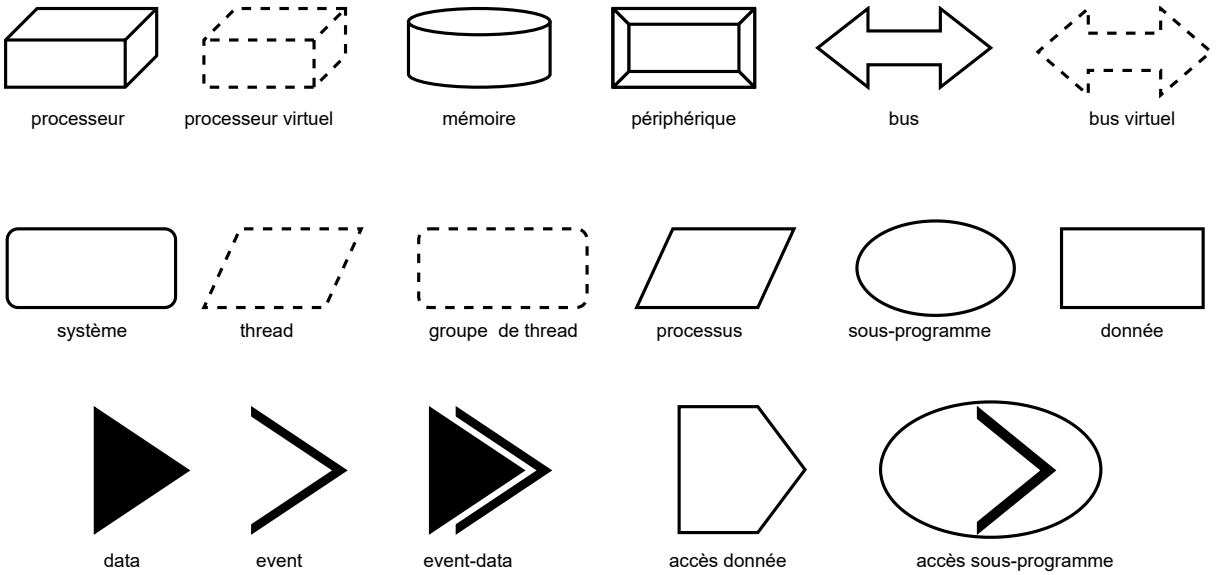

Figure 2.7: Composants graphiques d'AADL

2.4.5 UML MARTE

Le profil UML pour la modélisation et l'analyse des systèmes embarqués temps réel (UML MARTE) est une extension standardisée d'UML pour le développement dirigé par modèles de systèmes embarqués temps réel (RTES) [GS08]. Il a été développé par l'Object Management Group (OMG) et remplace le précédent profil UML pour l'ordonnancement, les performances et le temps (SPT).

Historique

UML MARTE a été développé par un consortium appelé ProMARTE, composé de membres de diverses entreprises et institutions. Le développement a été initié en raison des limites du SPT[1] (UML Profile for Schedulability, Performance and Time) en termes

[1]Un profil UML prédécesseur de MARTE qui avait une portée plus limitée et était basé sur d'anciennes versions d'UML

d'expressivité pour la modélisation des phénomènes RTES [GS08]. Un objectif supplémentaire était la mise à niveau du profil vers la nouvelle norme UML 2 et l'extension du champ d'application au-delà de l'analyse d'ordonnancement et de performance [GS08].

La proposition a été acceptée par l'OMG en 2007. La version 1.0 a été publiée après un processus de finalisation impliquant les fournisseurs d'outils fournissant des commentaires et résolvant les problèmes [GS08].

Utilisation

MARTE prend en charge à la fois la conception et l'analyse basées sur des modèles de RTES. Il fournit des concepts spécifiques au domaine pour la spécification, la modélisation architecturale, la conception détaillée et la vérification/validation [GS08].

Il peut être utilisé pour des tâches telles que :

- la conception de systèmes multitâches,
- la simulation de plateformes matérielles,
- l'analyse d'ordonnancement / performances basée sur un modèle,
- la modélisation architecturale de haut niveau.

MARTE est neutre sur le plan méthodologique et ne restreint pas l'utilisation d'UML. Il est souvent utilisé conjointement avec le profil SysML pour l'ingénierie des systèmes [GS08].

Composition

Le profil MARTE se compose de deux parties principales [GS08]:

- Modèle de conception MARTE : fournit des concepts spécifiques au domaine pour la modélisation des RTES.
- Modèle d'analyse MARTE : prend en charge l'annotation de modèles pour l'analyse telle que l'ordonnancement et les performances.

Il contient également des fondations avec des concepts de base comme la modélisation du temps et la modélisation de plateforme. Des constructions supplémentaires sont fournies dans les annexes.

La figure 2.8 montre la structure derrière MARTE, dont les modules clés sont [GS08; OMG23]:

- NFP - Modélisation de propriétés non fonctionnelles.
- Temps - Concepts temporels et phénomènes liés au temps.
- GCM - Modèle de composant général.

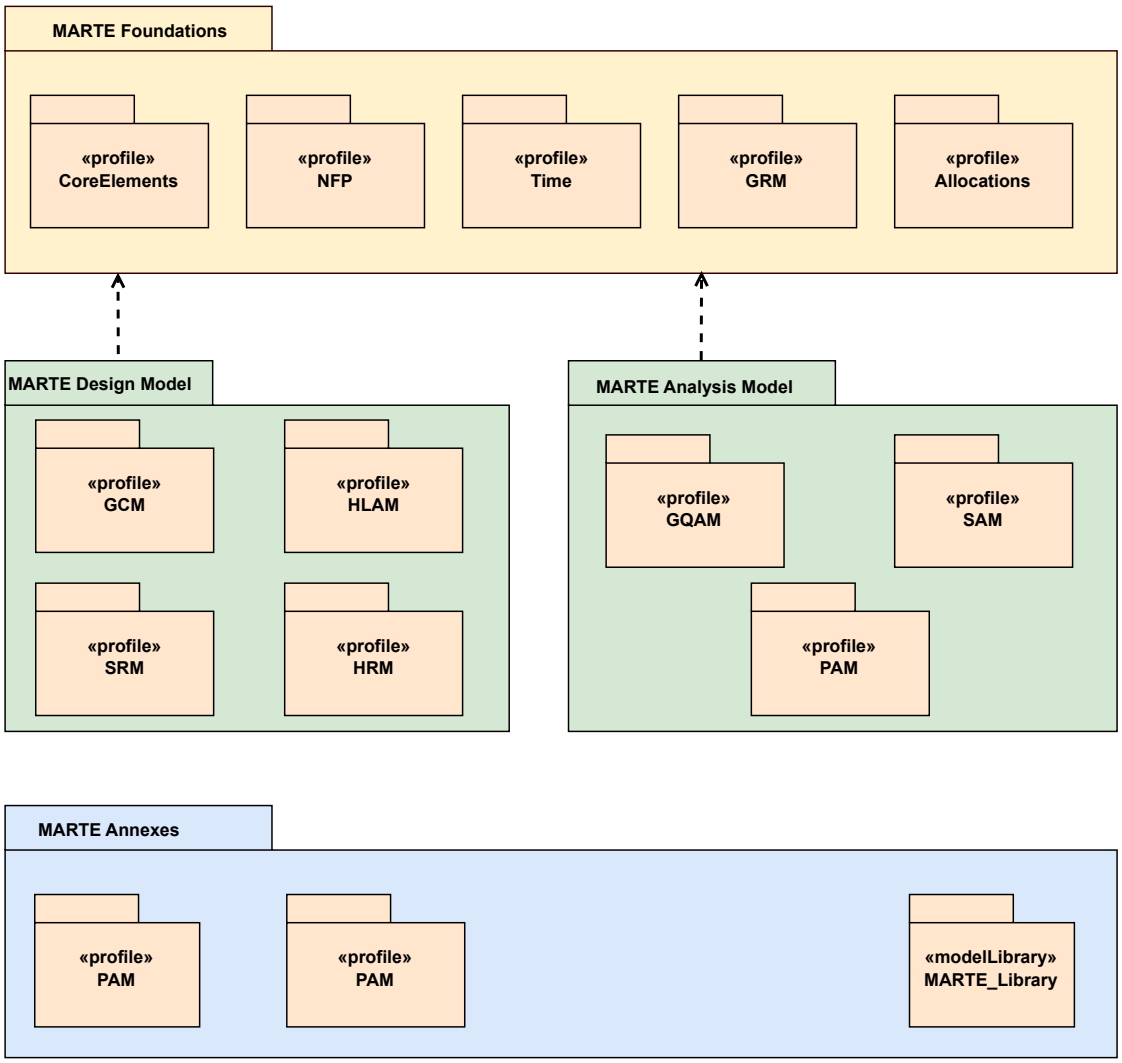

Figure 2.8: Architecture du profil MARTE

- GRM - Modélisation générique des ressources.
- SRM - Modélisation des ressources logicielles.
- HRM - Modélisation des ressources matérielles.
- Allocations - Mise en correspondance des applications avec les plateformes.
- GQAM - Modélisation d'analyse quantitative.
- SAM - Modélisation de l'analyse d'ordonnancement.
- PAM - Modélisation de l'analyse de performance.

Outils

MARTE peut être appliqué à l'aide d'outils de modélisation UML tels qu'IBM Rational Rhapsody, Papyrus, ArgoUML, MagicDraw, et bien d'autres. Il permet l'analyse basée

sur des modèles en s'intégrant à des outils externes via des transformations de modèles [GS08].

Des outils comme TTool fournissent un support pour la conversion de modèles et le retour d'information d'analyse. Des outils commerciaux prenant en charge MARTE sont proposés par des entreprises comme Esterel Technologies (SCADE), Telelogic (Rhapsody) et Artisan (Studio). L'OMG fournit également un tutoriel officiel MARTE avec des exemples.

Avantages et inconvénients

MARTE fournit des capacités significatives pour la modélisation des RTES :

- Il fournit des concepts spécifiques au domaine pour la modélisation des systèmes embarqués temps réel, ce qui rend la modélisation plus intuitive et efficace par rapport à l'UML pur.
- Il prend en charge plusieurs étapes de développement, y compris la spécification, la conception et l'analyse.
- Il permet l'analyse quantitative comme la modélisation de l'ordonnancement et des performances.
- Il permet la modélisation des aspects matériels et logiciels.
- Il s'aligne sur les normes industrielles comme SysML, AADL et EAST-ADL2.
- Il est indépendant de la méthodologie, et ne restreint pas l'utilisation d'UML.

En revanche, il présente également des limites en termes de complexité, d'outillage et de méthodologie qui doivent être traitées pour une adoption plus large par l'industrie. Des révisions complémentaires et des spécifications connexes peuvent aider à atténuer ces inconvénients, que nous listons ci-après.

- En tant qu'extension d'UML 2, courbe d'apprentissage abrupte pour les nouveaux utilisateurs.
- Manque de guidage et de méthodologie, les concepts MARTE peuvent être utilisés de manière incohérente.
- Support limité des outils actuellement et peu d'outils commerciaux le mettant pleinement en œuvre.
- Axé principalement sur la modélisation et l'analyse, manque de sémantique d'exécution.
- Ne remplace pas complètement SPT, référence encore certains concepts SPT.

2.4.6 Comparaison entre AADL et UML-MARTE

Le but de cette comparaison est de montrer les capacités des langages de modélisation de conception (Tableau 2.1). Pour cela, nous allons comparer les capacités de AADL et UML-MARTE sur différents critères tels que l'architecture système, la sémantique de l'information temporelle, la traçabilité et la phase d'utilisation dans le cycle de développement.

2.4.7 EAST-ADL

Electronic Architecture and Software Tools - Architecture Description Language (EAST-ADL) (EAST-ADL) est un langage de description d'architecture qui fournit une extension et un profil de SysML adapté à la modélisation des systèmes embarqués automobiles [Cue+08a]. Il permet de spécifier les exigences, les fonctions, les architectures logicielles et matérielles à un niveau d'abstraction plus élevé par rapport au code logiciel. EAST-ADL permet de capturer à la fois les propriétés fonctionnelles et non fonctionnelles des systèmes embarqués automobiles à différents niveaux d'abstraction. Il vise à gérer la complexité du développement des systèmes électroniques automobiles modernes en séparant les préoccupations et en permettant une modélisation et une analyse précoces.

2.4.8 EAST-ADL et AUTOSAR

EAST-ADL opère à un niveau d'abstraction plus élevé par rapport à AUTOSAR. Les modèles EAST-ADL représentent une spécification d'architecture fonctionnelle, tandis que les modèles AUTOSAR décrivent les détails de l'implémentation de l'architecture logicielle [Qur+10]. Le mapping des modèles fonctionnels EAST-ADL avec AUTOSAR permet le raffinement de l'architecture et la génération de code à partir de modèles de haut niveau. L'utilisation conjointe de EAST-ADL et d'AUTOSAR améliore le processus de développement en permettant une modélisation, une analyse, une vérification et une validation plus précoces. EAST-ADL complète AUTOSAR en fournissant la traçabilité des exigences aux modèles architecturaux et à la mise en œuvre [ATE10]. L'article présente un schéma de mise en correspondance entre les concepts EAST-ADL et AUTOSAR comme une étape vers la transformation automatique de modèles.

Types de composants EAST-ADL Le langage EAST-ADL fournit une modélisation de types de composants spécifiques à différents niveaux d'abstraction. Au niveau véhicule, seuls les composants de fonctionnalités peuvent être modélisés pour représenter les capacités du véhicule. Le niveau d'analyse comprend des composants de fonction et de périphérique pour capturer les fonctions logicielles et les interactions environnementales. Le niveau de conception permet la modélisation d'éléments architecturaux tels que capteurs, actionneurs, contrôleurs. Au niveau d'implémentation le plus bas, des composants AUTOSAR tels que fonctions logicielles de base, capteurs, ECU, etc. sont utilisés. EAST-ADL prend également en charge la modélisation de variabilité pour configurer des composants connexes en lignes de produits.

Tableau 2.1: Comparaison multicritère entre AADL et UML-MARTE

	AADL	**UML-MARTE**
Architecture système	L'approche basée sur les composants utilisée dans AADL permet d'élaborer une architecture très proche du système dans les cas pratiques. Néanmoins, les architectures modélisées via AADL sont figées.	UML-MARTE se concentre davantage sur des modèles qui peuvent être organisés sur des diagrammes séparés pour une modélisation incrémentale plutôt que l'architecture.
Sémantique de l'information temporelle / Phase d'utilisation dans le cycle de développement	La modélisation avec AADL nécessite de connaître plusieurs informations qui ne peuvent pas nécessairement être connues aux premières étapes de conception	UML-MARTE peut être utilisé à différentes étapes composant la phase de conception. De plus, les modèles UML-MARTE sont regroupés en deux couches : la couche applicative logique qui contient le modèle UML fonctionnel sans annotations, et la couche non-fonctionnelle composée des autres types de modèles comme le modèle de charge de travail, le modèle de plateforme, le modèle d'allocation, etc.
Traçabilité : capacité à récupérer les informations d'analyse à ajouter dans les modèles d'entrée	Actuellement, aucun ensemble standard de propriétés permet de conserver la traçabilité, mais il est possible d'ajouter cette fonctionnalité de manière ad-hoc	La traçabilité est garantie par les propriétés non-fonctionnelles d'UML-MARTE, mais reste complexe car elle nécessite de bien connaître le langage VSL
Sémantique de l'information temporelle	Les composants AADL ont une sémantique claire et immuable quelle que soit l'architecture conçue. De plus, dans le cas de l'utilisation d'OSATE, l'utilisation des différents éléments est vérifiée	En l'absence de sémantique standard, certains éléments ont des significations différentes selon la méthodologie suivie, le type de systèmes conçus, et les outils d'analyse qui seraient utilisés

AADL	EAST-ADL
Thread (thread d'exécution)	Fonction (fonction logicielle)
Données (stockage de données)	Fonctionnalité (capacité véhicule)
Périphérique (périphérique matériel externe)	Capteur (capteur matériel)
Processeur (plateforme d'exécution)	ECU (unité de contrôle électronique)

Tableau 2.2: Comparaison des types de composants

AADL	EAST-ADL
Port	Port de flux
Accès aux données	Port client-serveur
Accès bus	Broche matérielle
Sous-programme	
Paramètre	

Tableau 2.3: Comparaison des interfaces de composants

Interfaces de composants EAST-ADL EAST-ADL permet de modéliser différentes interfaces selon le niveau d'abstraction. Aux niveaux supérieurs, il prend en charge les ports de flux pour l'échange de données et les ports client-serveur spécifiant les opérations requises et fournies. Aux niveaux inférieurs, des interfaces de broches matérielles sont utilisées pour connecter les capteurs et actionneurs aux sources et destinations électriques. Le niveau d'implémentation utilise des interfaces AUTOSAR standardisées. Par exemple, un modèle d'environnement peut avoir des ports client-serveur pour fournir des données de capteur à d'autres fonctions.

2.4.9 Comparaison entre AADL et EAST-ADL

Les langages AADL et EAST-ADL sont tous deux des langages de description d'architecture utilisés pour modéliser les systèmes embarqués [JL11; FGH06]. Une différence clé entre eux est que AADL modélise les systèmes en utilisant des abstractions logicielles de bas niveau comme les threads et les processus, permettant une analyse détaillée et la génération de code [HF07]. En revanche, EAST-ADL opère à un niveau d'abstraction plus élevé, capturant les fonctionnalités système, les fonctions et les contraintes importantes pour l'industrie automobile [Cue+08c]. Les deux langages prennent en charge la modélisation de composants logiciels, de composants matériels et de configurations. Ils permettent également de spécifier des propriétés non fonctionnelles liées au timing et à la fiabilité, bien qu'EAST-ADL dispose d'un support plus explicite grâce à des packages dédiés [The10; AS-05]. Dans l'ensemble, alors que AADL est adapté à une analyse approfondie, EAST-ADL met davantage l'accent sur la compréhension et la communication des aspects de haut niveau du système [JL11]. Les tableaux 2.2 et 2.3 présentent respectivement une comparaison entre les types de composants et les interfaces de composants des deux langages.

Compte tenu de ces comparaisons, le choix du langage AADL comme base pour notre framework de déploiement se justifie pleinement. AADL permet de modéliser formellement les architectures logicielles et matérielles à différents niveaux d'abstraction, depuis les spécifications abstractives jusqu'aux implémentations détaillées. Ses concepts pré-

cisément définis facilitent l'analyse des propriétés critiques comme la performance et la sûreté. De plus, les sémantiques d'exécution d'AADL autorisent la génération de code d'interface et d'intégration système à partir des modèles. Cela permet de réduire les efforts de codage manuel et de garantir la cohérence entre conception et implémentation. Enfin, le métamodèle standardisé d'AADL assure l'interopérabilité entre les outils et améliore l'échange de modèles. L'ensemble de ces capacités répond parfaitement aux besoins de notre framework pour le déploiement fiable de systèmes critiques basé sur les modèles.

2.5 Conception des systèmes temps réel

Au fil des sections précédentes, nous avons vu diverses méthodologies, telles qu'AUTOSAR, ARCADIA ou MDA, dans lesquelles on peut trouver divers points communs dans la façon dont le processus de conception est mené. La conception des systèmes temps réel fait intervenir des modèles fonctionnels, matériels et opérationnels, illustrés dans la figure 2.9. L'objectif est d'obtenir à la fin du processus un modèle opérationnel correct et qui répond aux exigences fonctionnelles et extra-fonctionnelles.

2.5.1 Modèle fonctionnel

Le modèle fonctionnel se compose de fonctions, de leurs schémas d'activation, de leurs budgets de temps, de leurs relations et également de diverses contraintes qui doivent être respectées. Parmi ces contraintes, on peut trouver celles liées à l'ordonnancement (comme le délai de bout en bout) et celles liées aux aspects métier (par exemple, pour des raisons de sûreté, deux fonctions doivent être isolées).

Le comportement de la partie fonctionnelle peut être représenté sous la forme d'un graphe acyclique dirigé où les sommets correspondent à l'ensemble des fonctions du système et les arêtes correspondent à l'ensemble des interactions entre les fonctions.

2.5.2 Modèle matériel

Le modèle matériel se compose d'un ensemble de processeurs d'exécution, de leurs types de cœurs (par exemple, unicœur ou multicœur), de leurs types de fournisseurs/fréquences (par exemple, cœurs homogènes, uniformes ou hétérogènes) et d'un ensemble de bus réseau, de leur topologie, de leur bande passante et de leurs protocoles (comme CAN, AFDX, AVB, etc.).

2.5.3 Modèle opérationnel

La troisième partie est le modèle opérationnel qui est en fait le premier résultat du processus de déploiement. Le modèle opérationnel est le résultat du mapping (placement) des fonctions sur les tâches T (illustrées par les carrés oranges dans la figure 2.9), de l'allocation des tâches aux cœurs, du mapping des données sur les messages (illustrés par les flèches bleues), des échéances des tâches dérivées des délais fonctionnels de bout en bout, des schémas d'activation des tâches dérivés des activations et synchronisations des

fonctions, etc. De plus, le système d'exploitation temps réel (RTOS) est un choix crucial, car il a un impact sur l'ordonnancement des tâches et sur plusieurs propriétés, y compris les affectations de priorités, les coûts de préemption et l'accès aux ressources partagées.

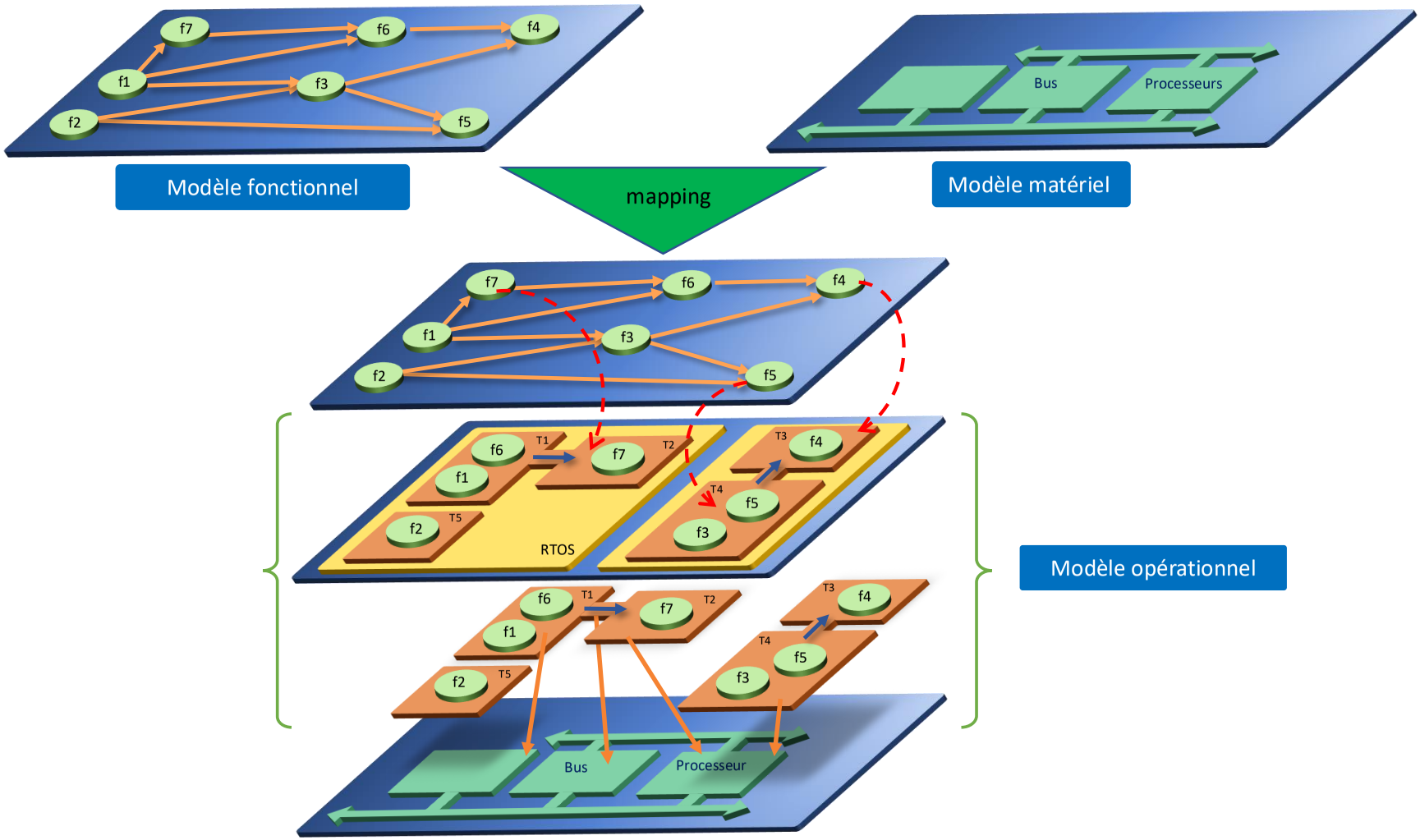

Figure 2.9: Aperçu des différentes parties d'un système temps réel

2.6 Conclusion

Dans ce chapitre, nous avons présenté les principales méthodologies et langages utilisés dans l'industrie pour la modélisation des architectures logicielles des systèmes embarqués temps réel critiques.

Nous avons vu que la séparation entre l'analyse fonctionnelle et le mapping sur l'architecture matérielle est un paradigme clé partagé par les approches telles qu'AUTOSAR, ARCADIA et MDA. Cette séparation des préoccupations apporte de nombreux avantages en termes de réutilisation, portabilité, évolutivité et vérifiabilité des architectures logicielles.

Les langages de description d'architecture comme AADL et EAST-ADL permettent une modélisation formelle des architectures logicielles/matérielles à différents niveaux d'abstraction. Leurs concepts précisément définis facilitent l'analyse rigoureuse des propriétés critiques comme la performance, la sûreté de fonctionnement et la sécurité.

Nous avons vu que AADL permet de modéliser formellement les architectures à différents niveaux d'abstraction et que ses sémantiques d'exécution autorisent la génération de code d'interface et d'intégration système. Cela motive son choix comme base pour notre framework de déploiement fiable de systèmes critiques.

CHAPITRE 3

Ordonnancement et validation des systèmes temps réel

3.1 Introduction

L'ordonnancement temps réel est le domaine scientifique qui veille à ce que tous les traitements concurrents chargés de contrôler une application critique s'exécutent sur une plateforme en respectant les échéances. Cet ordonnancement se déroule dans un contexte spécifique dépendant de l'architecture matérielle et logicielle ainsi que de l'exécutif temps réel (système d'exploitation ou programmation directe dite "bare-metal"). L'algorithme d'ordonnancement doit permettre la validation temporelle. Bien que cela ne soit pas le centre d'intérêt de la thèse, puisque nous nous plaçons à un endroit du cycle de vie où les estimations des durées d'exécution pire-cas des tâches ne sont pas connues, nous introduisons ici brièvement le domaine.

Les travaux de cette thèse ont, en effet, pour but de faciliter la validation temporelle ultérieure. Les applications cibles sont des applications industrielles s'exécutant sur un processeur, ordonnancées par des ordonnanceurs à priorités fixes aux tâches. Les problèmes d'ordonnançabilité sont donc présentés pour un modèle canonique de tâche simple, qui sert de référence dans l'ordonnancement temps réel. Nous détaillerons les paramètres clés de ce modèle, avant de présenter par la suite d'autres modèles de tâches plus élaborés. Le modèle canonique est fréquemment utilisé dans l'industrie grâce à sa simplicité d'analyse et sa capacité à modéliser un large éventail de systèmes. Cependant, des modèles plus complexes comme les tâches sporadiques, transactionnelles et à activations multiples offrent une expressivité accrue. Ces modèles seront évoqués dans les sections suivantes. Ils étendent le modèle canonique de base pour permettre une meilleure adéquation avec certains systèmes, au prix d'une complexité d'analyse plus importante.

3.2 Modèles de tâche

Les tâches temps réel sont des traitements exécutés de manière récurrente qui doivent être terminés dans le délai imparti. Les tâches exécutent les fonctions définies dans les étapes fonctionnelles du cycle de vie. Elles sont utilisées, par exemple, pour acquérir des données à partir de capteurs, calculer une commande et commander des actionneurs. Chaque activation d'une tâche est appelée une instance.

Les tâches temps réel peuvent être activées périodiquement, sporadiquement ou apériodiquement, avec des contraintes temporelles (échéances) à respecter. Pour guider l'ordonnancement, un modèle canonique a été défini pour caractériser les principales variables temporelles des tâches temps réel τ_i.

- Temps d'activation (r_i) : l'instant où la tâche est créée ou activée. Pour les tâches périodiques, il s'agit du temps d'activation de la première instance.
- Modèle d'activation/périodicité (T_i) : la période durant laquelle les instances de la tâche sont activées. Cela peut être périodique (activées à intervalles fixes), sporadique (activées avec un intervalle minimum entre les instances successives) ou apériodique (activées à des moments arbitraires). Pour les tâches apériodiques, le paramètre T_i n'est pas connu en général.
- Le temps d'exécution pire-cas (C_i) : le pire temps d'exécution nécessaire pour terminer une instance de la tâche, appelé WCET pour Worst-Case Execution Time.

- Échéance relative (D_i) : le délai dans lequel une instance de la tâche doit terminer son exécution après avoir été activée.

Dans la littérature, l'exécution d'une tâche τ_i est fréquemment représentée par une ligne de temps ou un diagramme de Gantt comme le montre la figure 3.1.

Chaque réveil de la tâche périodique déclenche une requête. Les dates de réveil successives de la tâche sont déterminées par la formule $r_k = r_0 + kT$, où r_0 représente le premier réveil et k représente le $k^{ème}$ réveil.

L'échéance absolue d d'une instance est calculée en ajoutant le délai critique D à la date d'activation de l'instance. Dépasser cette échéance entraîne une faute temporelle. L'échéance absolue d est calculée par rapport à son temps d'activation comme $d = r + D$.

La détermination de ces paramètres est souvent une étape importante de l'analyse et de la conception d'une application temps réel. Bien qu'il soit essentiel de prendre en compte les temps de commutation de tâches, les durées d'utilisation des primitives du système d'exploitation, les durées de prise en compte des interruptions et les durées d'exécution de l'ordonnanceur, ces éléments sont cependant soit négligés en général, soit supposés inclus dans le WCET des tâches.

Nous pouvons apporter les définitions supplémentaires suivantes pour une tâche temps réel sont :

- Taux d'utilisation : la fraction de temps processeur nécessaire à l'exécution de la tâche, dénotée $U_i = \frac{C_i}{T_i}$.
- Laxité/Marge dynamique (L_i) : le temps restant avant l'échéance moins le temps d'exécution restant.
- Temps de réponse (RT): le temps entre l'activation d'une instance et son achèvement. Il doit être au plus égal à D_i pour que l'échéance soit respectée.
- Gigue : variabilité des temps de réponse des instances.

Ces paramètres et propriétés décrivent fondamentalement le comportement temporel d'une tâche temps réel et sont essentiels pour analyser ses contraintes temporelles et son ordonnançabilité.

3.2.1 État d'une tâche

De manière générale, une tâche traverse quatre états une fois qu'elle est créée, comme illustré sur la figure 3.2.

Exécutée

Initialement, la tâche est créée et se prépare à être **exécutée**. Ensuite, elle entre en phase d'**exécution** une fois sélectionnée par l'ordonnanceur. À partir de l'état **en cours d'exécution**, la tâche termine son exécution, ou bien passe en mode **bloqué** ou **prêt**.

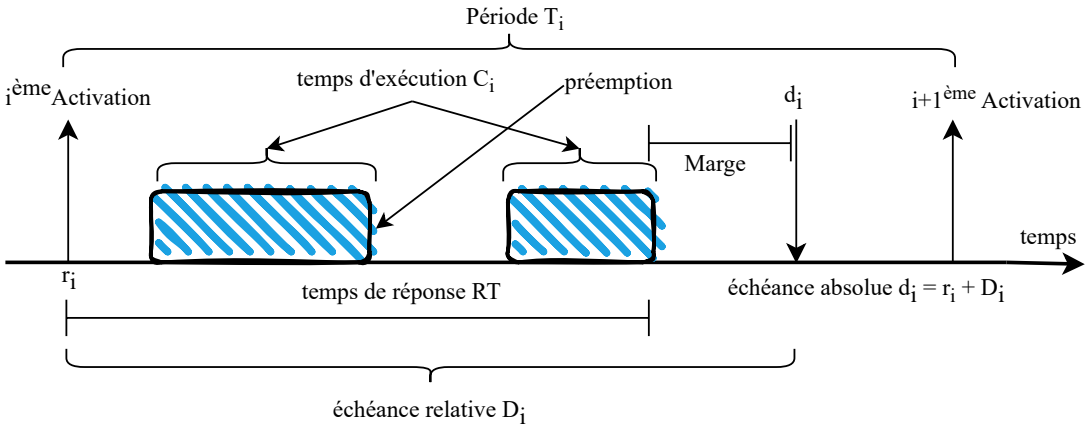

Figure 3.1: Modèle de tâches

Bloquée

Une tâche devient **bloquée** lorsqu'elle nécessite l'accès à une ressource déjà utilisée par d'autres tâches. Une tâche bloquée pourra reprendre son exécution une fois que la ressource sera disponible. Sinon, elle passe en mode **terminé** lorsqu'elle est supprimée.

Prête

Une tâche devient **prête** à être exécutée lorsqu'elle est **créée**, en attente de son tour d'exécution. Cependant, ce n'est pas le seul scénario possible. Une tâche peut également devenir prête lorsqu'elle est interrompue par une autre tâche plus prioritaire.

Terminée

À la fin de son exécution, la tâche passe en mode **terminé**. Une tâche qui est prête ou bloquée peut être supprimée avant d'avoir terminé son exécution pour passer à ce mode.

3.2.2 Hiérarchie des modèles de tâches

Au cours des dernières décennies, divers modèles de tâches ont été proposés pour représenter les charges de travail temps réel, augmentant progressivement en expressivité et en complexité d'analyse. Comme illustré à la figure 3.3, ces modèles forment une hiérarchie dans laquelle les modèles ultérieurs généralisent les précédents.

Le modèle de tâche périodique introduit par Liu et Layland [LL73b] en 1973 a été le premier modèle de tâche pour les systèmes temps réel. Les tâches périodiques libèrent des jobs de manière strictement périodique avec des échéances implicites. Liu et Layland ont déterminé un test suffisant d'ordonnançabilité qui consiste à vérifier que l'utilisation totale d'un système à n tâches est inférieure ou égale à $n(\sqrt[n]{2} - 1)$. Ce simple test d'utilisation a permis l'analyse des ensembles de tâches périodiques, mais la périodicité stricte limite l'expressivité.

Pour permettre des activations de jobs plus flexibles, le modèle de tâche sporadique [Mok83a] a été introduit en 1983, comme généralisation des tâches périodiques. Les tâches

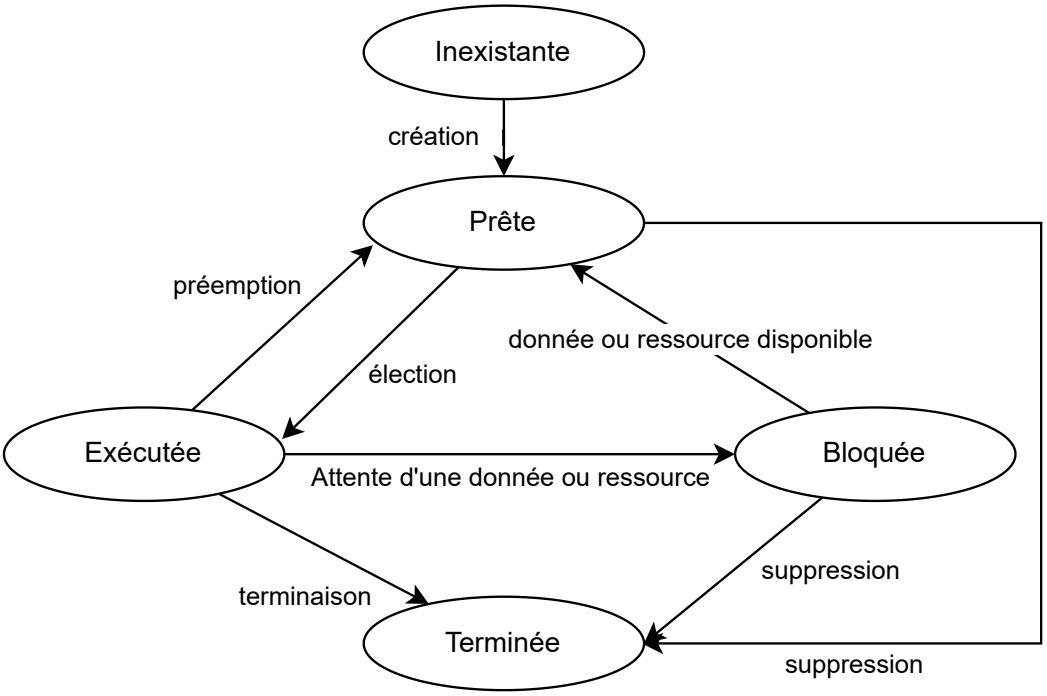

Figure 3.2: État d'une tâche [GRO+13]

sporadiques considèrent un délai minimum entre les activations, mais les activations peuvent se produire n'importe quand après cela. Le modèle périodique est donc un cas particulier du modèle sporadique, correspondant aux activations les plus resserrées possibles. Des échéances explicites ont également été ajoutées. Pour l'analyse de faisabilité, la fonction de demande a été introduite par Baruah et al. [Bar+90] en 1990. Elle compte la demande d'exécution maximale possible d'une tâche dans n'importe quel intervalle. Vérifier la demande pour toutes les tâches permet un test de faisabilité exact pour les tâches indépendantes préemptibles sur système monoprocesseur. La flexibilité supplémentaire entraîne un test plus complexe par rapport à la borne de Liu et Layland.

Le modèle à plusieurs trames de Mok et Chen [MC97] en 1997 permet à différents types de jobs, appelés trames, d'être activés par la même tâche. Cela généralise les tâches périodiques et sporadiques, et permet de modéliser des demandes d'exécution variables. Des tests suffisants d'ordonnançabilité ont été présentés pour l'ordonnancement par priorités fixes.

Le modèle de transactions est une autre généralisation du modèle sporadique. Dans ce modèle, les temps d'activation des transactions ne sont pas connus a priori [PH98; Rah+12].

Baruah et al. [Bar+99] ont introduit les tâches à trames multiples généralisées (GMF) en 1999 pour étendre davantage les tâches à trames multiples avec des délais entre les activations variables et des échéances pour les trames. Le concept de paire de demandes a été introduit pour une analyse de faisabilité exacte. La difficulté d'analyse augmente en raison de la variation accrue.

La modélisation des activations de jobs à embranchements a été rendue possible par le modèle d'embranchement récurrent de Baruah (RB) [Bar98b] en 1998, en utilisant des arbres. Il a été amélioré pour les tâches récurrentes temps réel (RRT) [Bar98a] en 1998, en utilisant des graphes acycliques dirigés (DAG) pour éviter les redondances.

La programmation dynamique a rendu l'analyse de faisabilité efficace. Les structures d'embranchement augmentent considérablement l'expressivité.

Le modèle GMF non cyclique [Moy+10] en 2010 supprime l'hypothèse de cycle de trame, permettant n'importe quel ordre de trame. La faisabilité utilise la programmation dynamique comme dans les tâches RRT. Les tâches RRT non cycliques [Bar10] en 2010 combinent encore plus l'expressivité de GMF et de RRT en utilisant des DAG. L'analyse unifiée a une complexité pseudo-polynomiale.

Le modèle temps réel à graphe dirigé (DRT) [Sti+11b], introduit par Stigge et al. en 2011, permet des graphes dirigés généraux, généralisant entièrement toutes les structures précédentes. Les techniques de tuples de demande permettent une analyse de faisabilité. Les tâches DRT étendues [Sti+11a] ajoutent des contraintes de synchronisation globales, marquant la frontière des tests pseudo-polynomiaux. L'expressivité basée sur les graphes est maximisée.

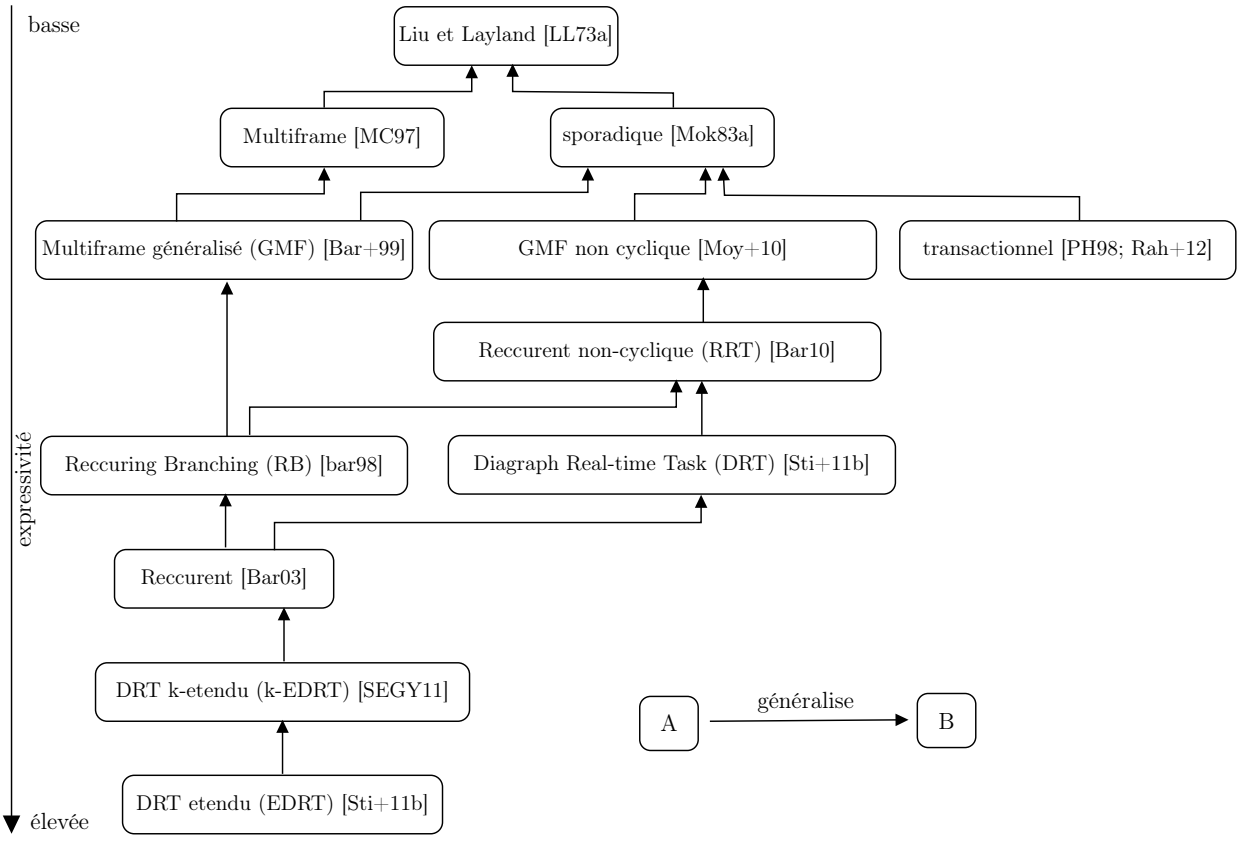

Figure 3.3: Hiérarchie des modèles de tâches [TGY20]

3.2.3 Classification des systèmes de tâches

Les systèmes de tâches temps réel peuvent être classés selon différents critères, notamment en fonction de leurs échéances et de leur capacité à être préemptés. Ces classifications permettent de mieux caractériser les ensembles de tâches et de déterminer les techniques d'analyse appropriées.

Échéances

Les tâches périodiques et sporadiques peuvent être classées en sous-catégories en fonction des caractéristiques de leurs échéances par rapport à la période d'activation. La littérature identifie trois catégories de tâches en fonction de leurs échéances :

- échéances implicites ou sur requête : $D_i = T_i$ pour chaque tâche t_i.
- échéances contraintes : $D_i \leq T_i$ pour chaque tâche t_i.
- échéances arbitraires : cela correspond au cas général où il n'existe aucune relation d'ordre imposée entre les échéances et les périodes des tâches.

Préemption

Une tâche préemptive est une tâche dont l'exécution peut être suspendue pour permettre à d'autres tâches plus prioritaires de s'exécuter. On distingue deux classes de systèmes de tâches selon ce critère :

- **Système de tâches préemptif :** si toutes les tâches sont préemptives et à tout moment.
- **Système de tâches non préemptif :** une ou plusieurs tâches ne peuvent pas être préemptées du tout.

Temps d'activation

Deux catégories de systèmes de tâches se démarquent en fonction du paramètre lié au temps d'activation : les tâches **concrètes** et les tâches **non concrètes**. Si les temps d'activation de toutes les tâches sont connus, on les qualifie de tâches **concrètes**, dans le cas contraire, on les considère comme **non concrètes**.

Lorsque l'ensemble de tâches est concret, si toutes les tâches sont déclenchées en même temps, alors on parle d'un ensemble de tâches concret et simultané ou **synchrone**. Sinon, l'ensemble de tâches est concret et **asynchrone**.

Durée d'exécution

La durée d'exécution est un élément indispensable pour les analyses d'ordonnançabilité. Elle peut être un intervalle borné entre le **BCET** (Best-Case Execution Time) et le **WCET** (Worst-Case Execution Time), comme elle peut prendre des valeurs probabilistes, etc. Souvent, c'est le **WCET** qui est utilisé au moment des analyses, dans le but de simplifier ces dernières, le BCET étant alors assumé comme nul.

Relations d'interdépendance

Selon les relations de précédence et l'existence de ressources partagées, un ensemble de tâches est catégorisé comme suit :

- **Ensemble de tâches indépendantes** : toutes les tâches sont indépendantes les unes des autres.
- **Ensemble de tâches dépendantes avec relations de précédence :** certaines tâches doivent attendre des messages ou des signaux de synchronisation provenant d'autres tâches.
- **Ensemble de tâches dépendantes** en raison de l'existence de ressources partagées.

Hiérarchie des modèles de tâches

Les systèmes de tâches peuvent être classés selon le modèle de tâche utilisé, qui définit les paramètres temporels et les activations des tâches. Les modèles vont du simple modèle périodique aux modèles plus élaborés comme les tâches sporadiques, détaillés dans la sous-section 3.2.3.

3.3 Ordonnancement monoprocesseur

Il existe deux principales catégories d'ordonnancement :

- **l'ordonnancement hors-ligne :** dans ce type d'ordonnancement, un ordonnanceur simplifié, appelé séquenceur, suit une table préétablie d'ordonnancement.
- **l'ordonnancement en ligne :** dans ce type d'ordonnancement, une stratégie d'affectation de priorité est utilisée pour permettre à l'ordonnanceur de choisir dynamiquement la tâche prête la plus prioritaire comme étant la prochaine tâche à exécuter.

3.3.1 Ordonnancement hors ligne

L'ordonnancement hors ligne implique la création préalable d'une table d'ordonnancement pour le système de tâches et au moment de la conception du système en respectant toutes les contraintes requises. Ce type d'ordonnancement est réalisable lorsque toutes les informations sur le système sont connues, y compris les dates de réveil des tâches. Par conséquent, les tâches doivent être périodiques.

3.3.2 Ordonnancement en ligne

l'ordonnancement en ligne d'un système monoprocesseur est principalement guidé par les priorités. Cette approche repose sur une politique d'affectation de priorités, où la tâche la plus prioritaire est exécutée en premier par l'ordonnanceur. Celui-ci est généralement informé des tâches actives et de leur état.

3.3.3 Classification et algorithmes d'ordonnancement

L'attribution des priorités peut être effectuée de manière ponctuelle ou de manière progressive au fil du temps lorsque le système est en service. Par conséquent, nous présentons la classification suivante.

Priorité fixe aux tâches

La priorité de chaque tâche est déterminée en fonction de ses paramètres statiques, tels que la période et l'échéance relative. L'attribution des priorités est effectuée avant le démarrage de l'application. Dans cette catégorie, on trouve l'algorithme Rate Monotonic (RM) [LL73b] proposé par Liu et Layland en 1973. L'algorithme affecte la priorité la plus élevée à la tâche ayant la plus petite période. En 1982 Leung [LW82] propose Deadline Monotonic (DM) qui affecte les priorités aux tâches on se basant sur leurs échéances. Globalement ces deux algorithmes ont la même philosophie : donner une priorité proportionnelle à la taille de la fenêtre d'exécution.

L'algorithme d'Audsley *Optimal Priority Assignement* (OPA) [Aud91] proposé en 1991, affecte des priorités de façon optimale en fonction des échéances des tâches également. Cependant, les contextes dans lesquels il s'applique nécessitent généralement un temps exponentiel pour vérifier une affectation de priorités.

Priorité fixe aux instances

Les instances au sein d'une tâche peuvent avoir des priorités différentes, mais chaque instance possède une seule priorité statique. Dans cette classe, on trouve l'algorithme Earliest Deadline First (EDF) proposé par Liu et Layland [LL73b] dans leur étude de 1973. Par la suite, ce travail a été repris par Horn [Hor74] et Dertouzos [Der74].

Priorité dynamique

Les priorités des instances peuvent changer entre leurs moments d'activation et leur fin d'exécution. Dans cette classe, on a l'algorithme d'ordonnancement à priorité dynamique Least Laxity First (LLF) [Mok83b] proposé par Mok en 1983. Les priorités des instances sont calculées en fonction de la proximité de leur échéance. Ainsi, l'instance la plus urgente est la plus prioritaire.

3.4 Ordonnancement multiprocesseur

L'ordonnancement multiprocesseur est guidé par la migration des tâches ou instances de tâches. Lors de la migration d'une tâche, il s'agit de préempter son exécution sur un processeur et de transférer son contexte vers un autre processeur afin de reprendre son exécution à partir du point de préemption. Le problème lié à ce type d'ordonnancement est un problème d'allocation dont l'objectif est de déterminer le processeur sur lequel la tâche ou l'instance d'une tâche va s'exécuter.

Les trois classes d'ordonnancement multiprocesseur selon Carpenter [Car+04] sont les suivantes :

3.4.1 Ordonnancement partitionné

Ce type d'ordonnancement est dit sans migration. Il consiste à allouer les tâches de façon statique sur les différents processeurs avant qu'elles soient exécutées. Les tâches s'exécuteront sur le même processeur durant toute leur vie. Une fois les tâches allouées, elles sont ordonnancées selon un ordonnancement monoprocesseur.

3.4.2 Ordonnancement semi-partitionné

Les tâches sont à migration restreinte, ou les tâches peuvent migrer d'un processeur vers un autre, mais seulement entre l'exécution de deux instances successives. Ainsi, une instance doit terminer son exécution sur le même processeur que là où elle a commencé.

3.4.3 Ordonnancement global

La migration dans ce cas est totale, c.-à-d. les tâches et les instances des tâches peuvent basculer d'un processeur à un autre sans restriction et à tout moment. L'ordonnanceur peut interrompre l'exécution d'une tâche sur un processeur à n'importe quel moment et reprendre son exécution sur un processeur différent.

3.5 Analyse d'ordonnançabilité et validation

3.5.1 Contexte d'analyse

Le contexte d'une analyse d'ordonnançabilité désigne l'ensemble d'hypothèses sur lesquelles repose l'analyse. Ainsi, un test d'ordonnançabilité est spécifique à un contexte donné. Un contexte englobe différents paramètres qui sont intrinsèquement liés aux tâches constituant le système étudié. Parmi ces paramètres, on retrouve le mode d'activation des tâches (périodique, sporadique ou apériodique), leurs réveils (simultanés ou différés), l'architecture matérielle du système (monoprocesseur, multiprocesseur ou distribué), ainsi que la politique d'ordonnancement utilisée, l'indépendance ou non des tâches, la possibilité ou non de préempter, etc.

Pour illustrer ces concepts, considérons un système composé d'un ensemble de tâches sporadiques indépendantes, préemptibles, avec un potentiel réveil simultané et à échéances contraintes. Ces tâches sont ordonnancées selon la politique Deadline Monotonic sur un système monoprocesseur. Dans ce contexte spécifique, l'analyse généralement utilisée repose sur le calcul du temps de réponse des tâches, qui est un test exact, de complexité pseudo-polynomiale.

3.5.2 Optimalité

Un système est fiablement ordonnancé par un algorithme d'ordonnancement si toutes les échéances sont respectées de l'instant 0 à l'infini.

Classes d'algorithmes	**échéances sur requête**	**échéances contraintes**	**échéances arbitraires**
priorité fixe aux tâches	Rate Monotonic (RM) [LL73b]	Deadline Monotonic (DM) [LW82]	Algorithme d'Audsley [Aud91]
priorité fixe aux instances	Earliest Deadline First (EDF) [LL73b]		
priorité dynamique	Least laxity First (LLF) [DM89]		

Tableau 3.1: Algorithmes optimaux en ordonnancement monoprocesseur

Un algorithme d'ordonnancement est considéré comme optimal pour une classe donnée d'algorithmes d'ordonnancement, et un contexte donné, s'il est capable d'ordonnancer fiablement tout système de ce contexte qui peut être fiablement ordonnancé par un algorithme de la classe considérée.

Pour les systèmes monoprocesseurs, dans le cas de tâches indépendantes et préemptibles, il existe un algorithme optimal pour chaque classe principale d'algorithmes donnée dans 3.3.2. Le tableau 3.1 donne les principaux algorithmes optimaux dans leurs contextes.

Pour les systèmes multiprocesseurs, Baruah [Bar+93] a proposé le premier algorithme appelé Pfair et a démontré son optimalité dans le contexte des tâches périodiques à échéance sur requête. L'algorithme utilise la migration totale et des propriétés dynamiques pour toutes les tâches. À l'exception de ce cas spécifique, Hong [HL92] a prouvé qu'il n'existe pas d'algorithme complètement en ligne optimal dans ce contexte.

3.5.3 Conditions d'ordonnançabilité

Les approches analytiques dans le domaine de l'ordonnancement se fondent sur la définition de conditions permettant de déterminer si un système de tâches est ordonnançable. Ces conditions reposent sur l'évaluation des critères temporels associés aux tâches du système. Selon les spécificités des systèmes auxquels elles s'appliquent, ces conditions peuvent être nécessaires, suffisantes ou à la fois nécessaires et suffisantes.

Condition suffisante

Un test d'ordonnançabilité est dit suffisant si tous les systèmes de tâches jugés ordonnançables selon le test sont effectivement ordonnançables.

Condition nécessaire

Si l'échec d'un test conduit inévitablement à un non-respect d'échéance à un moment donné pendant l'exécution du système, on peut alors le considérer comme une condition nécessaire.

Un test d'ordonnaçabilité à la fois suffisant et nécessaire est dit **exact**. Un test d'ordonnançabilité qui est uniquement suffisant est considéré comme **pessimiste**.

3.5.4 Hyperpériode

L'hyperpériode H d'un système de tâches périodiques $\tau_1, \tau_2, ..., \tau_n$ de périodes respectives $T_1, T_2, ..., T_n$ est définie comme le plus petit commun multiple (PPCM) de ces périodes :

$H = \text{ppcm}(T_1, T_2, ..., T_n)$

Elle représente l'intervalle après lequel le pattern d'activation des tâches se répète et permet de borner l'analyse d'ordonnançabilité du système.

3.5.5 Analyse d'ordonnançabilité

Le problème d'analyse d'ordonnançabilité consiste à déterminer si toutes les tâches dans τ, un ensemble de tâches, respecteront leurs échéances selon l'ordonnanceur donné. Comme l'analyse exacte de l'ordonnançabilité est NP-difficile au sens fort [ER08], nous désirons des tests suffisants en temps polynomial. Il existe plusieurs méthodes et techniques pour étudier l'ordonnançabilité à savoir l'utilisation processeur, la demande processeur, l'analyse du temps de réponse et la simulation.

Utilisation processeur

Le facteur d'utilisation (du processeur) est la fraction du temps processeur consacrée à l'exécution de l'ensemble des tâches. Il s'agit de calculer la fraction du temps processeur consacrée à l'exécution de l'ensemble des tâches. La formule utilisée pour calculer le facteur d'utilisation processeur est la suivante :

$$U = \sum_{i=1}^{n} \frac{C_i}{T_i} \tag{3.1}$$

Où n est le nombre total de tâches dans le système.

L'utilisation processeur doit être inférieure ou égale à un seuil pour avoir un système ayant une chance d'être ordonnançable dans le contexte des systèmes monoprocesseur, avec tâches périodiques ou sporadiques, concrètes, indépendantes, à échéances sur requête et préemptives ordonnancées selon la politique Rate Monotonic (RM). Ce seuil est :

$$n \times \left(\sqrt[n]{2} - 1 \right)$$

Le facteur d'utilisation processeur dans ce cas précis détermine une **condition suffisante** d'ordonnançabilité.

Demande processeur

Ce type d'analyse consiste à calculer la durée maximale des exécutions des tâches réveillées et terminées dans un intervalle de temps $[t_1, t_2]$ à l'aide de la fonction appelée *demand bound function* (DBF).

Analyse de temps de réponse (RTA)

Cette méthode consiste à calculer le temps de réponse pire cas (WCRT) de chaque tâche du système et le comparer aux échéances [JP86]. Si le temps de réponse d'une tâche dépasse l'échéance, le système est considéré non ordonnançable. Les approches basées sur le calcul du temps de réponse s'adressent à des systèmes complexes, à savoir des systèmes de tâches à échéances arbitraires, avec dépendance ou encore des systèmes distribués [TC94]. L'équation 3.2 [Aud+91] permet de calculer le temps de réponse d'une tâche τ_i pour un système préemptif, en considérant les tâches triées de la plus prioritaire à la moins prioritaire.

$$R_i = C_i + \sum_{j=1}^{i-1} \lceil \frac{R_i}{T_j} \rceil \cdot C_j \tag{3.2}$$

Une période d'activité au niveau de priorité i représente l'exécution contiguë des tâches avec une priorité $\geq i$. Elle commence et se termine à un instant inactif.

Audsley [Aud91] a montré que sous un ordonnancement à priorités fixes, le temps de réponse le plus long d'une tâche τ_i se produit pendant la plus longue période d'activité au niveau de priorité de τ_i.

Joseph et Pandya [JP86] ont prouvé que la fin de la plus longue période d'activité initiée à l'instant critique peut être calculée en temps pseudo-polynomial par une itération de point fixe :

$$w_i(0) = C_i$$
$$w_i(j+1) = C_i + \sum_{j \in hp(i)} \lceil w_i(j)/T_j \rceil C_j$$

où $hp(i)$ est l'ensemble des tâches avec une priorité supérieure à i. Le point fixe w_i^* donne le pire temps de réponse de la première instance de τ_i.

Si $w_i^* > T_i$, les instances ultérieures de τ_i peuvent également connaître de longs temps de réponse. Lehoczky [Leh90] a généralisé la RTA pour tenir compte de cela :

Algorithme 1 Calcul de WCRT

Entrée : Ensemble de tâches τ
Sortie : Pire temps de réponse R_i

1: $k \leftarrow 0$;
2: **Répéter**
3: $\quad k \leftarrow k + 1$;
4: $\quad w_i(k)(0) \leftarrow kC_i$;
5: $\quad w_i(k)(j+1) \leftarrow kC_i + \sum_{j \in hp(i)} \lceil w_i(k)(j)/T_j \rceil C_j$;
6: $\quad R_i(k) \leftarrow w_i(k)(j) - (k-1)T_i$;
7: **Tant que** $R_i(k) \leq T_i$
8: $R_i \leftarrow \max(R_i(1), \ldots, R_i(k))$;

Les équations itératives de la RTA fournissent un test d'ordonnançabilité exact et en temps pseudo-polynomial pour les ensembles de tâches avec des priorités statiques et sans dépendances inter-tâches [Aud91; JP86; Leh90].

L'analyse du temps de réponse fournit un moyen efficace d'analyser exactement l'ordonnançabilité des systèmes temps réel à priorités fixes, en modélisant la demande de charge de travail et l'offre du processeur. Elle est largement utilisée dans la théorie et les outils d'ordonnancement temps réel.

Simulation

La simulation est une technique courante pour étudier l'ordonnançabilité d'un système temps réel. Elle consiste à modéliser le système en fixant des paramètres comme la période T_i, le temps d'exécution C_i et le réveil r_i pour chaque tâche τ_i. L'ordonnanceur est ensuite simulé en répartissant les tâches selon l'algorithme d'ordonnancement choisi, et les temps de réponse sont mesurés pour vérifier que les échéances sont respectées.

Elle est utilisée pour reproduire le comportement d'un système et vérifier s'il est ordonnançable dans un intervalle de temps donné, appelé intervalle de simulation. Cet intervalle peut précisément être défini ou servir de borne supérieure à l'ordonnancement qui peut être reproduit de manière cyclique.

3.5.6 Notion de viabilité

En anglais “sustainability", a été introduit par Baruah et Burns en 2006 [BB06]. La notion de viabilité est étroitement liée à l'analyse d'ordonnançabilité et vise à garantir l'ordonnançabilité d'un système, même si son comportement réel est meilleur que celui prédit lors de sa conception, c'est-à-dire le comportement dans le pire cas.

Un test d'ordonnançabilité est considéré comme viable lorsque tout système ordonnançable par ce test reste ordonnançable même avec des paramètres améliorés par rapport aux paramètres pire cas des tâches.

La viabilité concerne les quatre paramètres, à savoir la durée d'exécution C, le réveil r, la période T, et enfin l'échéance relative D. Par conséquent, si un paramètre X parmi ces paramètres est amélioré et que le système reste toujours ordonnançable, alors le test est dit X-viable.

Viabilité de la simulation

Pour des tâches indépendantes préemptives sur monoprocesseur, la simulation est C-viable et T-viable [LL73a; Aud+93]. Avec des systèmes non concrets, elle est aussi r-viable [LL73a; Aud+93].

La simulation a une viabilité limitée dans certains contextes :

- Avec des contraintes de précédence, la simulation n'est pas C-viable. Elle reste viable si les précédences sont cohérentes avec les priorités [MS10].
- Avec des sections non préemptibles, la simulation n'est pas C-viable. Une analyse du temps de blocage additionnel est nécessaire.

Viabilité de l'analyse des temps de réponse (RTA)

L'analyse des temps de réponse (RTA) fournit des tests d'ordonnançabilité viables dans certaines conditions :

La RTA est viable :

- Suivant la pire durée : même si les tâches durent moins longtemps que prévu, le temps de réponse ne peut pas être plus long que le temps de réponse calculé par RTA.
- Suivant la date de réveil : l'instant critique où toutes les tâches sont activées simultanément est considéré. Par conséquent, s'il existe des tâches avec une date de réveil, le pire temps de réponse calculé par RTA est une borne supérieure.
- Suivant la période : si le délai entre deux activations successives est plus long que prévu, le temps de réponse ne peut être plus long que celui calculé par RTA.

3.5.7 Protocoles de gestion de ressources

L'ordonnancement des tâches avec contraintes de ressources soulève différents problèmes tels que les situations d'inversion de priorité et d'interblocage. L'inversion de priorité se produit lorsque la tâche la plus prioritaire τ_1 est bloquée en attendant une tâche moins prioritaire τ_2, qui à son tour attend une tâche τ_3 plus prioritaire que τ_2 mais moins prioritaire que τ_1. Une situation d'interblocage (deadlock en anglais) se produit lorsque deux tâches au moins se bloquent mutuellement, chacune attendant que l'autre termine son exécution, mais aucune des deux ne peut le faire.

Pour remédier à ces problèmes, plusieurs protocoles ont été définis.

Protocole à héritage de priorité

Dans ce protocole nommé en anglais *Priority Inheritance Protocol* (PIP), la tâche τ_1 qui détient la ressource critique acquiert la priorité d'une tâche τ_2 plus prioritaire bloquée par l'attente de celle-ci. Cette modification de priorité permet à la tâche τ_1 d'être ordonnancée à un niveau plus élevé que son niveau initial, dans le but de libérer rapidement la section critique, et ainsi minimiser l'attente de la tâche τ_2. Cependant, il est important de noter que le protocole d'héritage de priorité ne prévient pas les interblocages [SRL90], et qu'une tâche peut se retrouver plusieurs fois en attente d'une section critique lorsqu'elle accède à plusieurs ressources critiques.

Protocole à priorité plafonnée

Dans ce protocole (en anglais *Priority Ceiling Protocol* (PCP)), chaque ressource critique se voit initialement attribuer une priorité plafond qui correspond à celle de la tâche de plus haute priorité pouvant demander son accès. Cette priorité agit comme un seuil, puisqu'une tâche ne peut entrer en section critique que si le sémaphore requis est libre, mais aussi si sa priorité est supérieure au plafond système. Ce plafond est la priorité plafond maximale de l'ensemble des ressources en cours d'utilisation

Dans la version immédiate de PCP, lorsqu'une tâche τ_i accède à la ressource, elle adopte immédiatement la priorité plafond, qu'elle conservera durant sa section critique [SRL90].

Les deux versions de PCP garantissent non seulement l'absence d'interblocage, mais aussi qu'une tâche ne peut être bloquée, durant son exécution, que par une section critique de priorité inférieure.

Protocole de ressources basé sur une pile

En anglais Stack Resource Policy (SRP)[Bak91], ce protocole est une variante du protocole (PCP) qui intègre plusieurs extensions visant à améliorer l'ordonnancement et à gérer différents scénarios de ressources. La SRP étend le PCP pour gérer des ressources multiunités telles que les sémaphores binaires et les verrous lecteurs-écrivains, ainsi que les schémas de priorité dynamiques tel que EDF avec des "niveaux de préemption" statiques. De plus, elle introduit le concept de partage de l'espace de la pile d'exécution entre les tâches.

La SRP intègre trois extensions clés : la gestion des ressources multiunités, les schémas de priorité dynamiques et le partage de la pile. Ces extensions peuvent être appliquées indépendamment ou ensemble. La SRP suppose que chaque tâche peut nécessiter l'utilisation d'une pile partagée, ce qui lui permet d'éviter les changements de contexte inutiles et de simplifier sa mise en œuvre en utilisant une structure de pile.

La SRP fournit un résultat d'ordonnancement spécifique pour l'ordonnancement EDF avec la SRP. Ce résultat est prouvé être plus précis que celui établi pour EDF avec une version dynamique du PCP. Cela implique que la SRP offre de meilleures garanties d'ordonnancement et de meilleures limites sur l'inversion de priorité dans le cadre de l'ordonnancement EDF.

3.6 Outils d'analyse basés sur les modèles

Lors de l'utilisation de solutions de conception (modèles), la conception doit être analysée pour vérifier que les exigences temporelles d'un système temps réel sont respectées. Effectuer manuellement les analyses d'ordonnancement est fastidieux et sujet aux erreurs, la raison pour laquelle plusieurs outils open source académiques et commerciaux ont été développés.

3.6.1 Cheddar

Cheddar [Sin+04] est une boîte à outils d'analyse, d'ordonnançabilité et de simulation écrite en Ada pour les systèmes temps réel. Cheddar peut manipuler des modèles AADL sous certaines conditions, cependant le modèle ne doit pas être complexe, et il doit être enrichi par plusieurs propriétés adhoc seulement comprises par Cheddar. Les systèmes de base à analyser peuvent être décrits par le langage de conception Cheddar. Le formalisme d'entrée et de sortie de Cheddar est basé sur XML. De plus, pour faciliter la modélisation, Cheddar fournit une interface graphique.

Bien que Cheddar prenne en charge un grand nombre de tests d'ordonnançabilité, il existe des cas où les tests existants ne correspondent pas aux caractéristiques d'un système donné. Pour ces cas, Cheddar offre la possibilité de définir de nouveaux tests d'analyse. Cette tâche nécessite de bien comprendre l'architecture interne de l'analyseur Cheddar.

3.6.2 MAST

MAST (Modeling and Analysis Suite for Real-Time Applications) [MAS] est une suite d'outils d'analyse d'ordonnançabilité basée sur des modèles développée en Ada pour les applications temps réel. MAST a ses propres métamodèles basés sur XML, un métamodèle d'entrée pour créer les modèles à analyser, et un métamodèle de sortie pour les résultats d'analyse.

MAST fournit un environnement utilisateur facilitant la création de modèles, un visualiseur de résultats pour afficher les résultats de l'analyse, et également une interface graphique pour choisir un test d'analyse. En plus de supporter les systèmes temps réel de base, MAST supporte également les architectures distribuées. De plus, le langage de conception MAST permet de concevoir le noyau des threads en proposant les éléments *Opération*. Le type d'une opération peut être simple, composite ou enveloppante. Les éléments qui composent la dernière version des métamodèles MAST sont très proches de ceux de SAM, un package du profil UML-MARTE. Même si MAST est open source, l'extension de MAST pour ajouter de nouveaux tests nécessite de programmer de nouveaux tests à partir de zéro.

3.6.3 ASIIST

ASIIST (Application Specific Input/Output Integration Support Tool) [Nam+09] est un outil qui supporte les modèles de système spécifiés en AADL pour effectuer de nombreux types d'analyses temps réel. ASIIST a été développé comme un plug-in Eclipse [Ecl23] et fonctionne avec l'éditeur AADL OSATE [OSA23]. ASIIST permet aussi l'analyse des architectures Modulaires Avioniques Intégrées (IMA). En effectuant l'analyse d'ordonnançabilité pour ces applications, ASIIST peut détecter les effets survenus en raison du trafic d'entrée/sortie existant dans les systèmes. ASIIST résout les équations d'analyse grâce à Mathematica [Wol23].

3.6.4 Rt-Druid

RT-Druid [Gai+07] est un outil d'analyse et un environnement de conception implémenté comme un plug-in Eclipse [Ecl23] avec une interface graphique. Bien que RT-Druid soit orienté vers les applications automobiles (il prend en charge la norme OSEK/VDX [Gro23a]), il est possible de l'utiliser dans n'importe quel autre domaine de systèmes temps réel. Les systèmes conçus via l'outil RT-Druid sont définis par une partie application, et une partie plate-forme (les ressources de la plate-forme, matérielles et logicielles). Ensuite, RT-Druid capture le placement des composants fonctionnels du système aux threads concurrents, et fournit des tests d'analyse d'ordonnançabilité.

3.7 État de l'art sur les méthodes de déploiement des systèmes temps réel

Le fonctionnement idéal d'un système permettrait l'exécution continue et instantanée de toutes les fonctions nécessaires, répondant instantanément aux exigences fonctionnelles. Cependant, chaque fonction nécessite un temps de calcul non nul. Par conséquent, les architectes système doivent stratégiquement déployer les fonctions à travers les tâches logicielles et les ressources matérielles, en ayant une certaine connaissance de l'architecture cible. En informatique, l'attribution de fonctions à des tâches distribue la charge de calcul, et impose un rythme d'activation aux fonctions contenues dans des tâches. Nous appelons le mapping des fonctions sur des tâches et les tâches sur le matériel "déploiement" (voir figure 3.4). Le déploiement pose des défis pour la correction du système. De mauvais choix de conception peuvent ne pas respecter les délais, produisant des systèmes peu performants. La performance peut être vue comme la possibilité pour des chaînes fonctionnelles de présenter des délais de bout en bout plus ou moins longs pour une architecture matérielle donnée. La détection tardive de mauvais déploiements augmente exponentiellement les coûts, car plus les problèmes apparaissent tard, plus le prix à payer pour les rectifier est élevé. Un déploiement soigneux des fonctions à travers les tâches et le matériel est crucial lors de la phase de conception pour éviter les problèmes de correction et de performance qui deviennent de plus en plus coûteux à résoudre tout au long des étapes du cycle de développement.

Au cours des deux dernières décennies, plusieurs méthodes ont été proposées pour résoudre le problème de synthèse de logiciels à partir d'une spécification fonctionnelle initiale. Dans [BLD05], les auteurs ont proposé une approche ad-hoc pour cartographier les fonctions sur une architecture monoprocesseur ciblant les tâches en utilisant la politique d'ordonnancement Earliest Deadline First (EDF). Leur solution consiste à créer un modèle de charge de travail spécifique avec la mesure des propriétés temporelles à partir d'un modèle fonctionnel. Supportant l'exclusion mutuelle, les blocs fonctionnels partagent des ressources logiques. En outre, les ressources partagées sont synchronisées en utilisant le Priority Ceiling Protocol (PCP) ou la politique de pile (Stack Resource Policy).

Saksena et al. [SKW00] ont proposé une approche automatique pour la synthèse d'un modèle de tâches à partir d'un modèle de conception avec des exigences de temps de bout en bout. L'approche proposée s'adresse à une plateforme monoprocesseur où les tâches sont exécutées selon la politique Deadline Monotonic (DM). L'allocation est basée sur une technique de branch and bound. Les auteurs cherchent à minimiser l'inversion de priorité et à réduire le nombre de tâches ainsi que les communications inter-tâches. L'événement de déclenchement des tâches peut être synchrone ou asynchrone. La technique d'analyse d'ordonnançabilité adoptée est basée sur le temps de réponse.

Kodase et al. [KWS03] ont proposé une méthode de transformation générique basée sur des modèles pour générer un modèle de déploiement à partir d'un modèle fonctionnel. Le processus de transformation est le suivant : tout d'abord, ils identifient les transactions dans le modèle fonctionnel. Ensuite, ils attribuent des priorités aux actions des composants et, enfin, ils affectent les actions aux tâches. Les priorités sont attribuées de manière statique aux tâches selon la politique d'ordonnancement Deadline Montonic. En outre, la communication entre les composants est asynchrone. Les auteurs s'adressent à une plateforme matérielle avec un processeur à un seul cœur avec un ordonnanceur DM.

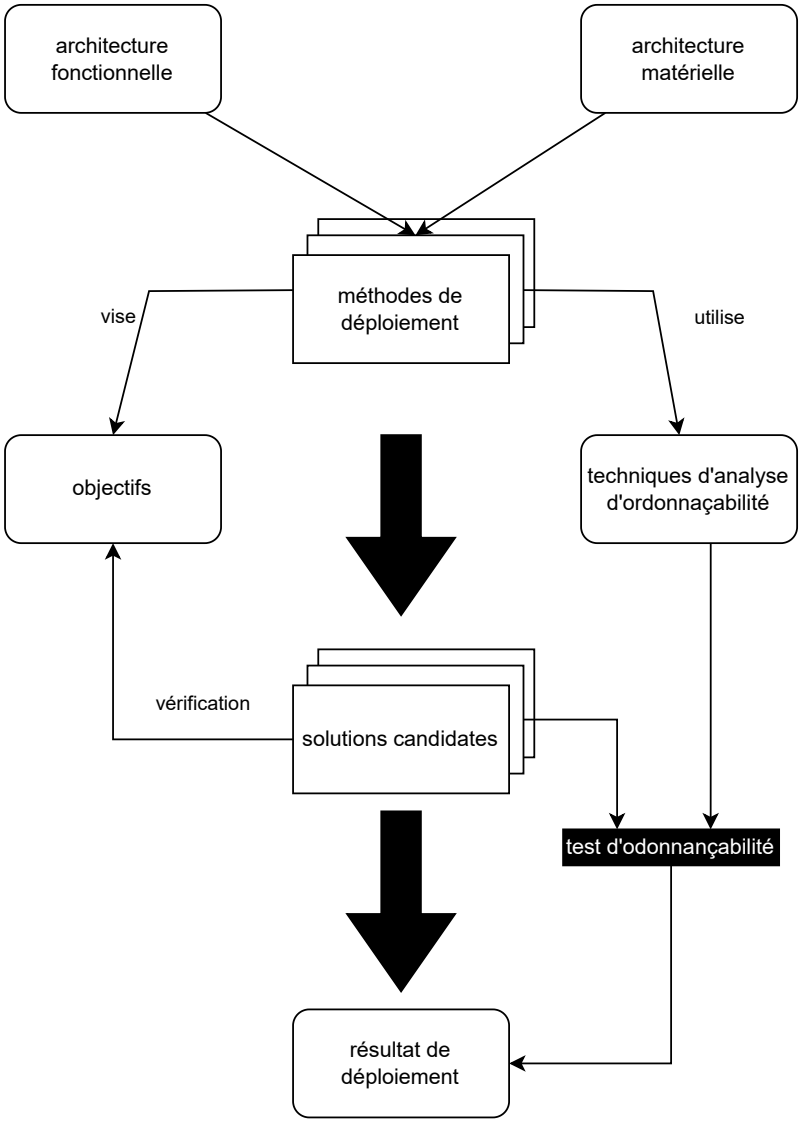

Figure 3.4: Aperçu de la phase déploiement

Ils visent à obtenir une implémentation avec une utilisation élevée du processeur et des surcharges d'exécution faibles. Les tâches qui partagent la mémoire sont synchronisées avec un temps de blocage.

Mehiaoui et al. [Meh+13] ont étudié les architectures distribuées avec des nœuds d'exécution hétérogènes suivant une politique d'ordonnancement à priorité fixe aux tâches. Ils ont utilisé une formulation en deux étapes basée sur la programmation linéaire en nombres entiers. La première étape consiste à placer les fonctions sur les nœuds d'exécution, puis les affecter aux tâches dans une seconde étape. À la fin de la seconde étape, les tâches résultantes sont ordonnancées. Le problème est formulé sous la forme d'optimisation sous contraintes linéaires d'une fonction objectif pouvant être résolue à l'aide d'un solveur MILP. Par conséquent, les auteurs ont abordé deux objectifs : minimiser en même temps le nombre de tâches et le temps de réponse de bout en bout. Les auteurs ont adopté l'analyse du temps de réponse comme test d'ordonnançabilité pour valider les solutions candidates [TC94]. Une solution optimale peut résulter de la résolution d'un problème formulé à l'aide de la programmation linéaire. Cependant, la solution ne se généralise pas bien lorsqu'un problème a un grand nombre de possibilités et plusieurs contraintes.

Par conséquent, des approches approximatives, telles que les heuristiques, peuvent être utilisées à la place. Mehiaoui et al. [Meh+18] ont abordé le problème en deux étapes en utilisant un algorithme génétique afin de surmonter les limitations liées à l'utilisation de techniques de programmation linéaire.

Bertout et al. [BFO14] ont proposé une méthode pour regrouper certaines tâches tout en s'assurant que le système de tâches reste ordonnançable. Leur objectif était de minimiser le nombre de tâches pour réduire les coûts liés à la commutation de contexte en explorant un espace de recherche bien délimité. Le système de tâches étudié se compose de tâches périodiques et indépendantes. Ils visent des architectures ayant un seul processeur suivant les politiques d'ordonnancement DM et EDF. Le problème étant démontré NP-difficile, il est traité à l'aide d'une heuristique de type Best-First Search (BFS). Le test d'ordonnançabilité utilisé pour valider les solutions candidates est principalement l'analyse du temps de réponse [JP86].

Saidi et al. [Sai+15] ont proposé une méthode permettant de faire le mapping dans le contexte d'AUTOSAR en proposant une formulation en programmation linéaire en nombres entiers (ILP [1]). Il s'agit d'affecter l'ensemble des fonctions (ou Runnable) dépendantes à un ensemble de tâches périodiques à échéance sur requête, et les tâches à plusieurs cœurs de processeur qui partagent une mémoire commune. Les auteurs supposent une approche d'ordonnancement partitionné, où les tâches sont assignées statiquement aux cœurs. Chaque cœur dispose de sa propre instance de système d'exploitation, exécutant un ensemble de tâches qui peuvent communiquer à la fois intra et inter-cœurs. Ce travail vise à minimiser les communications inter-coeur de processeur et équilibrer la charge processeur.

Les auteurs de [Bou+18] ont ciblé les architectures monoprocesseurs avec une politique d'ordonnancement Rate Monotonic (RM). Un modèle fonctionnel avec des fonctions périodiques et indépendantes a été considéré. Ils ont proposé une méthode d'optimisation à deux objectifs, réduire le nombre de préemptions et de maximiser la laxité (la quantité de temps entre le temps de réponse pire cas d'une tâche et son échéance relative). Le problème a été résolu avec un algorithme méta-heuristique appelé Pareto Archived Evolutionary Algorithm. Pour vérifier l'ordonnançabilité, un test basé sur la simulation est effectué sur chaque solution candidate donnée.

Lakhdhar et al. [Lak+19] ont proposé une formulation en programmation linéaire mixte en nombres entiers (MILP [2]) pour minimiser le nombre de tâches tout en optimisant individuellement un seul objectif, que ce soit en minimisant le temps de réponse, en minimisant l'allocation de mémoire ou en minimisant la consommation d'énergie, selon le choix du concepteur. Les auteurs ciblent une plateforme matérielle monoprocesseur avec un ordonnanceur à priorité fixe. Les tâches sont dépendantes et communiquent via la mémoire partagée en utilisant un mécanisme de sémaphore avec le protocole Priority Ceiling (PCP).

Le Nabec et al. [LHB18] ont proposé une approche formelle pour composer différentes stratégies de déploiement afin d'optimiser à la fois les critères temps réel et de qualité de conception. Leur framework QuaRTOS-DSE permet de combiner différentes techniques de clustering, dont Joined Late Activation (JLA), et de les appliquer successivement sur différentes parties du système. Des évaluations basées sur plusieurs critères comme le

[1] Integer Linear Programming
[2] Mixed-Integer Linear Programming

nombre de tâches, le temps de réponse, le mélange des catégories de fonctions, etc. sont ensuite effectuées. Leur approche facilite ainsi l'exploration de l'espace de conception en prenant avantage des différentes techniques de déploiement pour répondre à des critères multiples. Des études de cas sur un système de protection de relais et un système de freinage ont démontré l'intérêt de combiner JLA et d'autres algorithmes pour optimiser à la fois les performances temps réel et la qualité de conception.

Nous avons classé les méthodes dans le tableau 3.2 selon les critères suivants : le contexte de déploiement considéré, les principaux objectifs d'optimisation, l'approche algorithmique proposée pour résoudre le problème ainsi que le test d'ordonnançabilité utilisé. Ce tableau permettra au lecteur d'avoir une vue d'ensemble des travaux existants dans ce domaine et de mieux situer notre contribution par rapport à l'état de l'art.

Tableau 3.2: Classification des travaux existants

Travaux	Contexte	Objectifs	Solution	Ordonnançabilité
[SKW00]	**modèle fonctionnel :** périodique, sporadique, dépendent **Synchronisation :** synchrone et asynchrone **politique d'ordonnancement :** deadline monotonic **plateforme matérielle :** monoprocesseur **protocole de synchronisation :** PCP [Bur99]	minimiser l'inversion de priorité et le nombre de threads et les communications inter-tâches	algorithme de recherche branch and bound	temps de réponse [JP86]
[KWS03]	**modèle fonctionnel :** périodique, apériodique **Synchronisation :** communications asynchrones **politique d'ordonnancement :** Deadline-monotonic **plateforme matérielle :** monoprocesseur **protocole de synchronisation :** temps de blocage	maximiser l'utilisation du porcesseur et réduire les surcoûts d'exéction	méta-heuristique (simulated annealing)	non mentionné
[BLD05]	**modèle fonctionnel :** périodique et sporadique dépendent **Synchronisation :** OR seulement **politique d'ordonnancement :** EDF **plateforme matérielle :** monoprocesseur **protocole de synchronisation :** PCP/SRP [Bak91]	réduire l'inversion de priorité et le changement de contexte	solution adhoc (heuristique)	demande processeur [BMR90]
[BFO14]	**modèle fonctionnel :** tâches périodiques et indépendantes **Synchronisation :** non mentionné **politique d'ordonnancement :** RM/EDF **plateforme matérielle :** monoprocesseur **protocole de synchronisation :**non mentionné	minimiser le nombre de tâches minimiser les surcoûts liés aux changements de contexte	Best-First Search heuristique	temps de réponse [JP86]

[Sai+15]	**modèle fonctionnel :** tâches périodiques et dépendantes **Synchronisation :** synchrone **politique d'ordonnancement :** Rate Monotonic **plateforme matérielle :** multicœurs ECUs **protocole de synchronisation :** sémaphore sur les structures de données partagées	minimiser les communications inter-coeur de processeur et équilibrer la charge processeur	formulation ILP	borne d'utilisation de Rate Monotonic
[Bou+18]	**modèle fonctionnel :** périodique et indépendant **Synchronisation :** synchrone **politique d'ordonnancement :** Rate Monotonic **plateforme matérielle :** monoprocesseur **protocole de synchronisation :** PCP	minimiser le nombre de préemptions ou de changements de contexte ou le nombre de tâches ou leur marge ou le temps d'exécution maximal ou le temps de blocage ou le nombre de ressources partagées	Stratégie d'Évolution Archivée de Pareto (PAES)	simulation sur hyperpériode [LM80]
[Lak+19]	**modèle de fonctionnel:** Periodique(WCET) **Synchronisation :** non mentioné **politique d'ordonnancement :** Rate monotonic **plateforme matérielle :** monoprocesseur **protocole de synchronisation :** PCP	réduire le nombre de tâches ou améliorer la consommation d'énergie ou l'allocation de mémoire ou le temps de réponse.	formulation MILP	temps de réponse [JP86]
[Meh+18]	**modèle fonctionnel :** périodique, sporadique **Synchronisation :** OU, ET **politique d'ordonnancement :** fixed priority **plateforme matérielle :** Plateforme matérielle distribuée avec des nœuds hétérogènes **protocole de synchronisation :** PCP	Minimiser à la fois le nombre de tâches et le temps de réponse de bout en bout pire cas des événements externes	formulation MILP	temps de réponse [TC94]

3.8 Conclusion

Ce chapitre a présenté les concepts clés et les modèles de tâches relatifs à l'ordonnancement temps réel. Les modèles de tâches définissent les paramètres temporels des tâches et leur activation, allant des modèles simples périodiques aux modèles plus complexes à échéances arbitraires.

Les algorithmes d'ordonnancement monoprocesseur et multiprocesseur ont été classifiés selon leur stratégie d'attribution des priorités. Les algorithmes optimaux ont été identifiés pour différents contextes d'ordonnancement.

L'analyse d'ordonnançabilité permet de déterminer si un ensemble de tâches respectera ses échéances. Plusieurs techniques ont été présentées : calcul de l'utilisation processeur, analyse de la demande processeur, analyse du pire temps de réponse et simulation. Les conditions de viabilité ont également été discutées.

La gestion des ressources partagées soulève des problèmes d'inversion de priorité et d'interblocage, contre lesquels des protocoles spécifiques ont été développés, notamment le protocole d'héritage de priorité, le protocole de priorité plafonnée et le protocole de ressources basé sur une pile.

Cet état de l'art sur l'ordonnancement temps réel pose les bases théoriques et les outils d'analyse nécessaires pour appréhender la validation temporelle des systèmes critiques. Les chapitres suivants dépendent des concepts présentés dans ce chapitre.

Nous avons également défini le déploiement et avons présenté un état de l'art sur les travaux en relation avec les méthodes proposées pour optimiser le déploiement des systèmes embarqués temps réel critiques. Nous pouvons constater que les auteurs considèrent connues les durées d'exécution des fonctions. Cependant, nous avons pu voir dans les méthodes employées que la phase de déploiement s'effectue bien avant que les fonctions soient mappées, exécutées et mesurées sur une architecture matérielle spécifique. Ce point central est crucial dans cette thèse, puisque nous avons souhaité proposer une méthode de déploiement dont les entrées sont conformes à ce que le concepteur connaît au moment du mapping, et donc, en particulier, pas les durées d'exécution des fonctions.

CHAPITRE 4

RYM : Une approche basée sur les rythmes pour le pré-déploiement des RTES critiques

4.1 Introduction

L'un des avantages majeurs de cette méthode réside dans sa capacité à réduire le nombre de tâches et à minimiser les temps de réponse bout en bout dans les scénarios les plus pessimistes en matière de chaîne fonctionnelle. En optant pour la méthode des rythmes, les concepteurs peuvent l'appliquer à la main et peuvent en créer des variantes.

Pour rendre encore plus accessible le processus de conception, nous introduisons également un outil logiciel à base des modèles. Celui-ci offre aux concepteurs système la possibilité de spécifier tant les composants fonctionnels que non fonctionnels du système, tout en générant automatiquement une configuration de pré-déploiement conforme à la méthode des rythmes. Cette approche intégrée offre une solution complète et efficace pour la conception de systèmes embarqués temps réel performants et fiables.

4.2 Principe général

Nous supposons que nous avons en entrée un graphe de fonctions, type graphe flots de données, sans cycle, formant un DAG. C'est assez commun dans les méthodes de spécification conception présentées dans les chapitres précédents. Les nœuds d'entrée dans le DAG sont typiquement des fonctions d'acquisition capteur, réseau, ou utilisateur, alors que les nœuds de sortie sont en général des commandes envoyées aux actionneurs, réseaux, ou consoles. Chaque chaîne fonctionnelle allant d'un nœud d'entrée à un nœud de sortie est munie d'une échéance de bout en bout, issue des exigences non fonctionnelles.

Ces fonctions doivent être mappées sur des tâches. Nous considérons des systèmes de tâches conformes au profil Ravenscar [Bur99], en particulier, chaque tâche possède un seul mode d'activation et les communications entre tâches sont asynchrones (par tableau noir) ou synchrones faiblement couplées (par boîte aux lettres). Nous ne considérons pas, comme dans [Bur99], l'utilisation de communication fortement couplée type rendez-vous. Nous allons chercher à regrouper les fonctions par rythme d'activation (une tâche correspond à un unique rythme), en parcourant les chemins du DAG, qui forment des chaînes fonctionnelles. Afin de faciliter le raisonnement sur les rythmes d'entrée, nous pourrons être amenés à représenter sur le DAG de fonctions les nœuds d'entrée (capteurs, réseaux, dispositif d'entrée, etc.).

Bien que nous ne connaissions pas la durée des fonctions, nous devons faire une unique hypothèse, celle que chaque tâche sera exécutée en respectant son échéance. Cette hypothèse ne pourra être vérifiée qu'une fois le système effectivement déployé et implémenté sur la plateforme cible. Nous considérons qu'il est de la responsabilité du concepteur de savoir si le système de tâches affecté à un processeur a de fortes chances de ne pas trop augmenter sa charge.

Puisque nous parcourons les chaînes fonctionnelles, nous commençons par considérer les tâches d'entrée. Elles sont soit asynchrones (typiquement tâche d'acquisition sur un capteur analogique), soit synchrones par rapport à la donnée attendue qui va activer la tâche (i.e., capteur I2C, série, USB, réseau Ethernet, etc.).

Une tâche de scrutation périodique est asynchrone. Elle est activée indépendamment des dates de changement des données pouvant être lues sur le capteur. Par conséquent, le

délai maximal entre le changement de la donnée entrante scrutée et la fin de l'exécution de la tâche (traitement de cette donnée) est, d'après notre unique hypothèse, $T + D$, c'est-à-dire la période d'activation de la tâche plus son échéance relative (voir figure 4.1).

Nous considérons des tâches avec des échéances contraintes (c'est-à-dire des échéances des tâches inférieures ou égales aux périodes) qui sont très courantes dans les applications industrielles. En conséquence, le pire délai est borné par $2T$ ($T + D \leq 2T$). Considérer $D = T$ est aussi guidé par le fait qu'il est plus simple de valider des tâches à échéances sur requête que des tâches à échéances contraintes.

En revanche, les tâches sporadiques et apériodiques sont synchrones (déclenchées) sur une donnée ou événement d'entrée et sont activées à leur arrivée. Par conséquent, le pire délai entre l'arrivée d'une donnée (ou événement) entrante et la fin de l'exécution de la tâche est égal à $D \leq T$ comme le montre la figure 4.2.

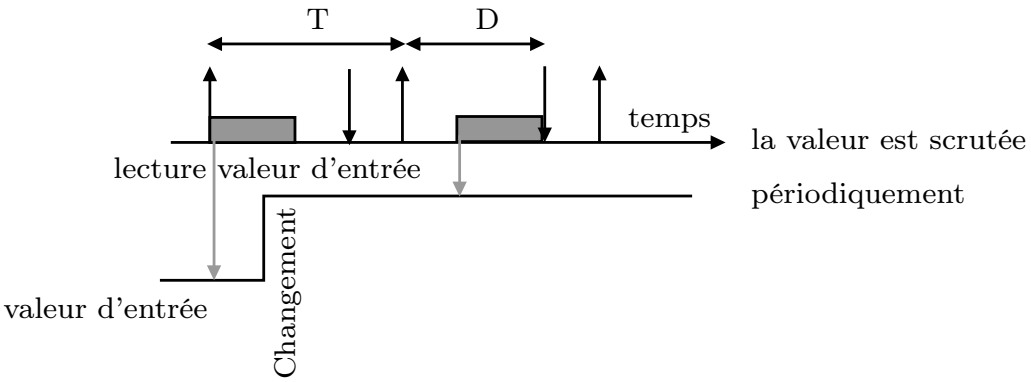

Figure 4.1: Pire délai entre changement de l'entrée et terminaison d'une tâche dans le cas asynchrone : le délai entre activation de la tâche et changement est arbitrairement petit

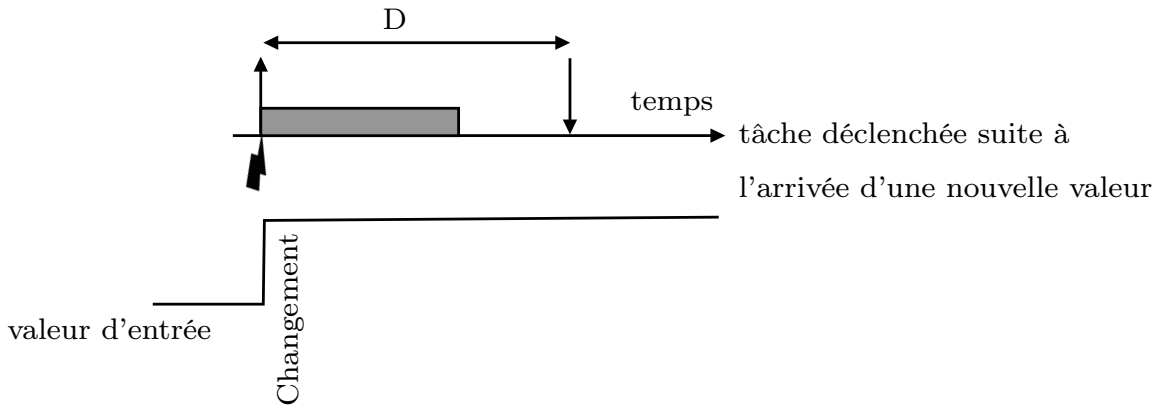

Figure 4.2: Pire délai entre changement de l'entrée et terminaison d'une tâche dans le cas synchrone

4.2.1 Identification des chaînes fonctionnelles

Chaque fonction d'entrée de chaîne fonctionnelle commence un ou plusieurs flots de données. Par exemple, le diagramme présenté dans la figure 4.3 comprend trois chaînes fonctionnelles de bout en bout :

- C1 : du capteur *E1* à la fonction *F2*,
- C2 : du capteur *E2* à la fonction *F2*,
- C3 : du capteur *E2* à la fonction *F5*.

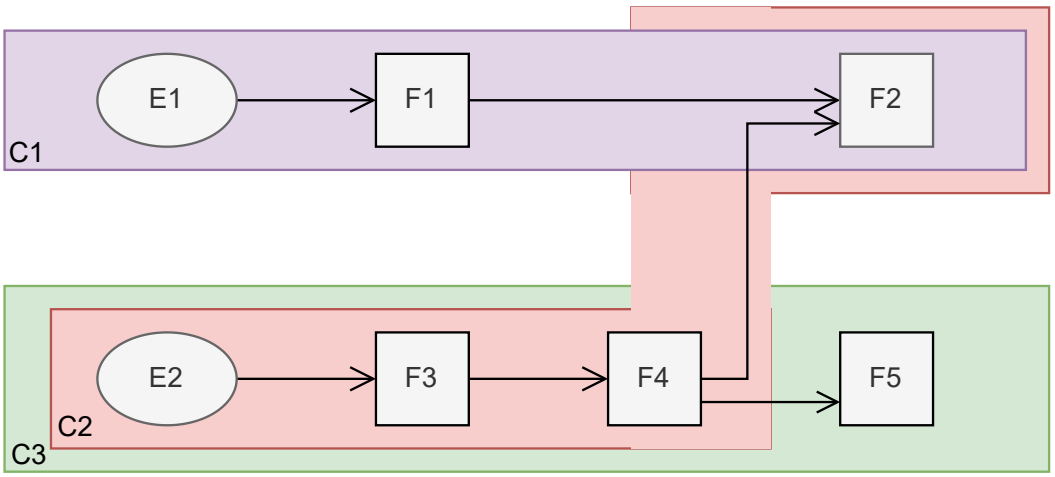

Figure 4.3: Identification des chaînes fonctionnelles

4.2.2 Acquisition de données

Le choix des rythmes (la fréquence ou la cadence à laquelle certains événements se produisent) source implique de déterminer la manière dont la première tâche de la chaîne fonctionnelle est activée. Elle peut être périodique, sporadique, apériodique, etc. D'une façon générale, les rythmes apériodiques ne permettent pas (sauf à utiliser un mécanisme spécifique de test d'admission) de faire du temps réel dur, nous ne les considérons donc pas. Dans le cas des tâches périodiques, la période est égale à $T = T^{min} = T^{max}$, tandis que pour les tâches sporadiques, elle est comprise entre $T = T^{min}$ et $+\infty$. Le rythme de la tâche d'acquisition dépend de la propriété T^{max} de la fonction qui collecte les données du capteur. Si T^{max} n'est pas infini, la tâche est activée périodiquement, comme le montre la figure 4.4(a). Lorsque T^{max} est infini, la tâche est activée de manière sporadique comme le montre la figure 4.4(b).

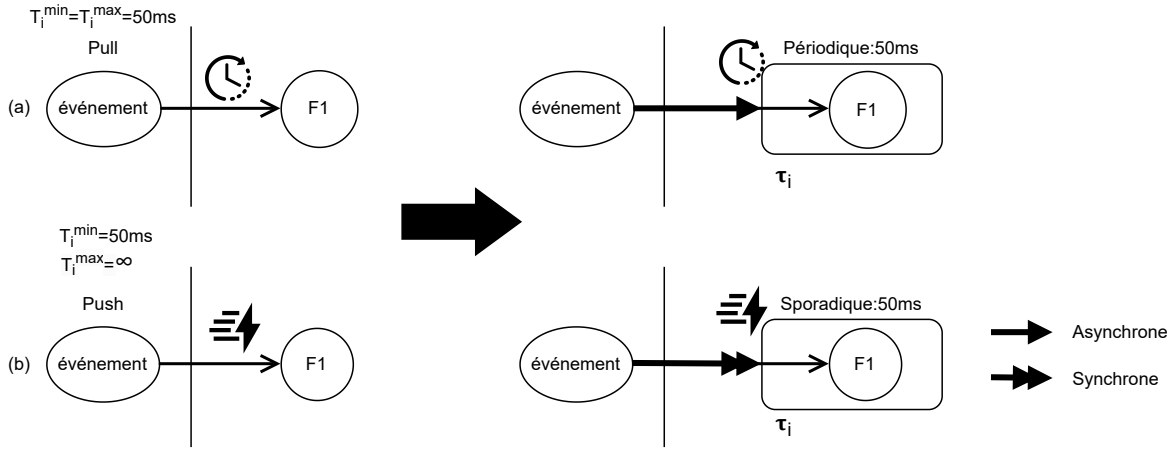

Figure 4.4: Création de tâche d'acquisition

4.2.3 Placement d'une fonction ayant un seul flot de données entrant

Chaque flot de données sera considéré, du plus urgent au moins urgent. Lorsque l'on considère un flot, une fois que la première fonction est placée dans une tâche, nous passons à sa fonction successeur qui appartient à la même chaîne fonctionnelle. Si la fonction considérée appartient à une seule chaîne fonctionnelle, elle peut être ajoutée séquentiellement au même rythme que l'acquisition. Il n'y a aucune raison de le faire plus rapidement ou plus lentement. Par conséquent, en suivant la méthode des rythmes, la fonction est placée séquentiellement dans la même tâche que son prédécesseur. Cette règle peut être maintenue tant que l'ensemble des fonctions appartient à une seule chaîne fonctionnelle, comme illustré dans la figure 4.5.

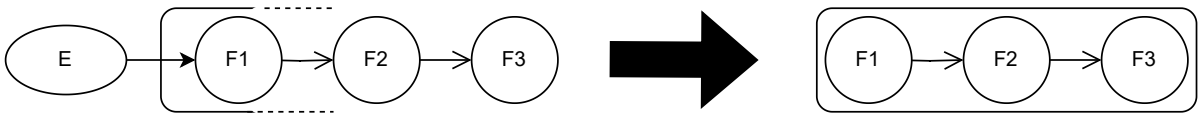

Figure 4.5: Exécution séquentielle de 3 fonctions dans la même tâche.

Suivant notre unique hypothèse (temps de réponse inférieur ou égal à la période minimale), mettre des fonctions dans la même tâche ne présente que des avantages :

- Placer la fonction successeur dans une tâche synchrone ou asynchrone augmenterait le pire délai de bout en bout. Ainsi, placer F2 dans la même tâche que F1 sur la figure 4.5 ne coûte rien sur le pire délai de bout en bout. Cependant, la placer dans une seconde tâche synchrone déclenchée par la tâche exécutant F1 ajoute dans le délai de bout en bout un délai maximal égal à la plus petite période de cette nouvelle tâche. Pire encore, créer une tâche de période T exécutant F2, venant lire de façon asynchrone la donnée produite par F1 ajoute $2T$ dans le délai de bout en bout.

- La mémoire utilisée par deux tâches, l'une exécutant F1, l'autre F2, est supérieure (taille de la pile à prévoir) à celle utilisée par une seule tâche exécutant séquentiellement F1 et F2.
- Les conséquences de l'ajout d'une tâche supplémentaire sur le système d'exploitation, que cela soit sur l'utilisation processeur (préemptions) ou mémoire (BCP, éléments de gestion d'une tâche, taille des files d'attente, etc.), constitue un surcoût. De plus, une donnée échangée par deux tâches apparaît typiquement trois fois : localement dans la tâche émettrice, localement dans la tâche réceptrice, et dans l'outil utilisé pour la communication (variable globale partagée pour un tableau noir, ou buffer pour une boîte aux lettres, plus sémaphores garantissant l'exclusion mutuelle, et outil de synchronisation).
- Un système est d'autant plus complexe et difficile à valider, notamment durant les tests d'intégration, qu'il possède plus de tâches.

4.2.4 Communications inter-tâches

En ce qui concerne la communication potentielle entre deux tâches selon la méthode des rythmes, si deux tâches s'exécutent à des rythmes différents $R1$ et $R2$, une communication asynchrone doit être établie comme le montre la figure 4.6 (a). Cela a pour effet d'ajouter un délai de bout en bout de $2T$. En effet, l'asynchronisme coûte une période sur ce délai (voir figure 4.1), et le fait d'avoir une seconde tâche coûte un délai critique (que nous considérons comme une période, les tâches étant à échéance sur requête). En revanche, si les deux rythmes peuvent être identiques, une communication synchrone est établie (voir figure 4.6 (b)).

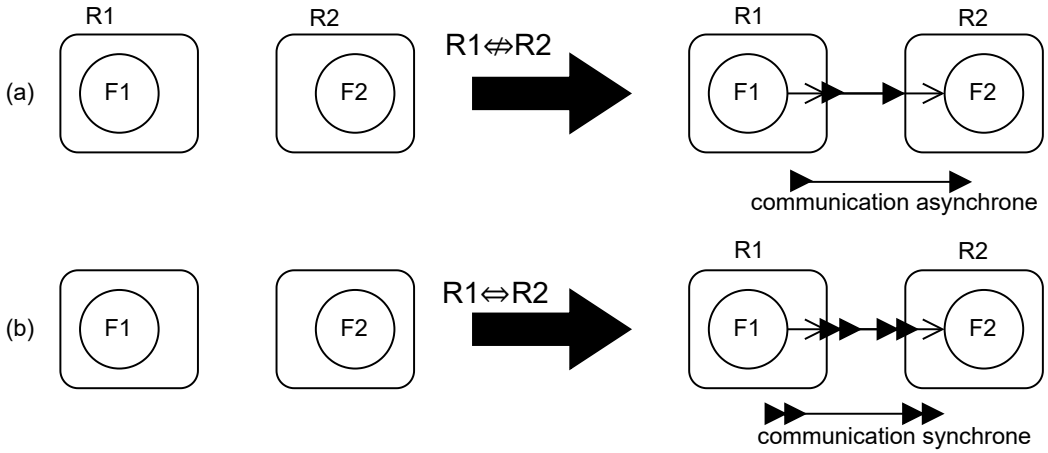

Figure 4.6: Différents types de communication.

4.2.5 Allocation d'une fonction avec plusieurs flots de données entrants

Dans cette situation, nous nous trouvons dans la même configuration que celle illustrée dans la figure 4.7, qui représente une convergence de deux chaînes fonctionnelles différentes dans la même fonction (F3 dans ce cas). Nous avons donc trois choix possibles :

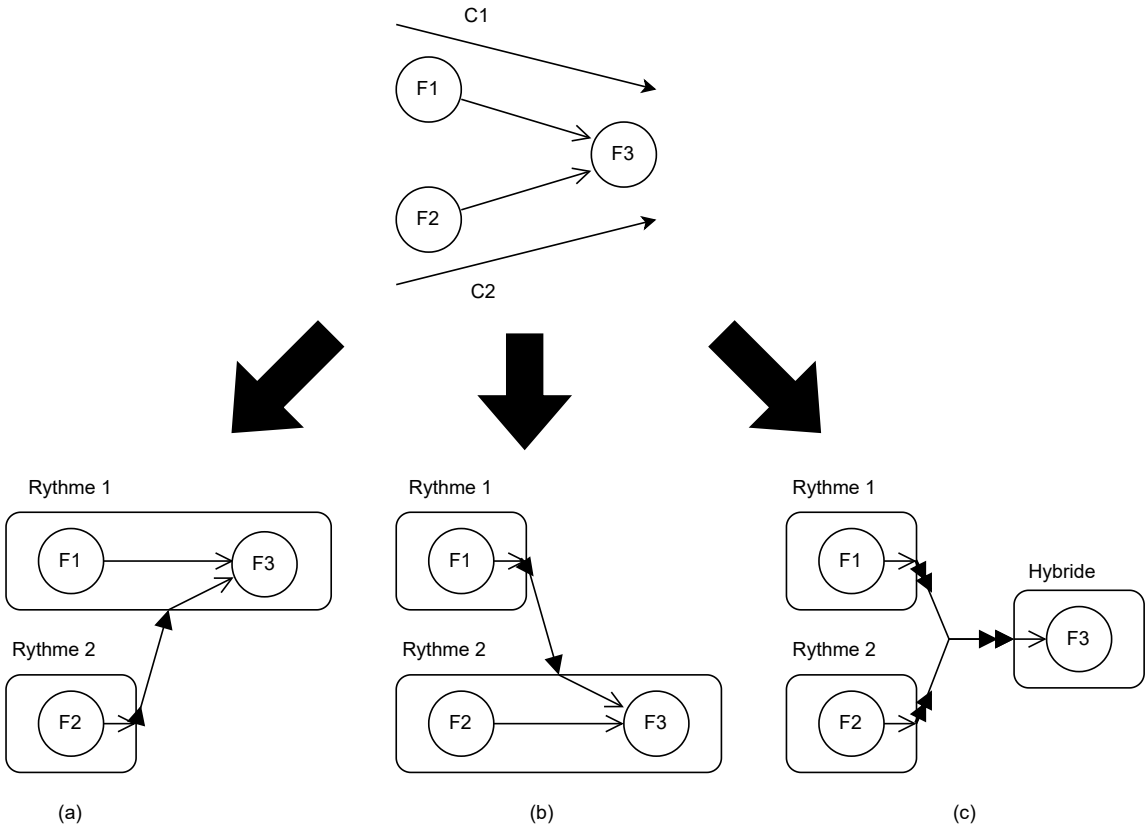

Figure 4.7: Convergence de plusieurs flots de données en une seule fonction.

- L'intégrer dans la tâche activée selon le rythme 1 (comme indiqué dans la partie (a) de la figure 4.7). La chaîne F1-F3 n'a aucun surcoût dans le délai de bout en bout par rapport à la sous-chaîne F1. Par contre, si la tâche de rythme 1 a une période comprise entre T^{min} et T^{max}, la chaîne F2-F3 a un surcoût de $T^{max} + T^{min}$ par rapport à la sous-chaîne F2. Notons donc que si le rythme 1 est sporadique (i.e., $T^{max} = +\infty$), cette construction est interdite, puisqu'elle ajoute un délai pire cas infini sur la chaîne F2-F3.

- L'intégrer dans la tâche activée selon le rythme 2 (comme indiqué dans la partie (b) de la figure 4.7). Dans les deux cas, les données sont communiquées de manière asynchrone entre les tâches, en raison des rythmes différents des tâches qui communiquent entre elles. Le même argument que précédemment s'applique : cette construction est interdite si le rythme 2 est à une période maximale infinie (i.e., si le rythme est sporadique).

- Le troisième choix consiste à faire un compromis pour les deux flots de données en plaçant la fonction dans une tâche hybride (comme indiqué dans la partie (c) de la figure 4.7). Cette tâche sera activée de deux manières différentes selon deux types d'activation, soit une activation 1-sur-N (sémantique d'activation en OU), soit une activation N-sur-N (sémantique d'activation en ET). Dans le cas OU, chaque chaîne, F1-F3 et F2-F3 voit son délai de bout en bout allongé par la période minimale de la tâche hybride. Dans le cas ET, ce délai est plus complexe à obtenir, puisqu'il dépend en plus du délai maximal entre l'arrivée de chaque type d'activation provenant de rythme 1 et de rythme 2.

– Sémantique d'activation 1-sur-N : également appelée sémantique d'activation en OU, cela signifie que la présence d'un seul événement est suffisante pour déclencher la tâche, représentée par un seul port qui rassemble tous les flots d'entrée synchrones (voir figure 4.8 (a)).
– Sémantique d'activation N-sur-N : également appelée sémantique d'activation en ET. La tâche ne peut pas être déclenchée tant que tous les événements d'entrée ne sont pas arrivés. Cela est symbolisé par autant de ports synchrones que de flot de données entrants, comme illustré dans la partie (b) de la figure 4.8.

Nous utiliserons donc toujours la sémantique de l'activation en *OU*.

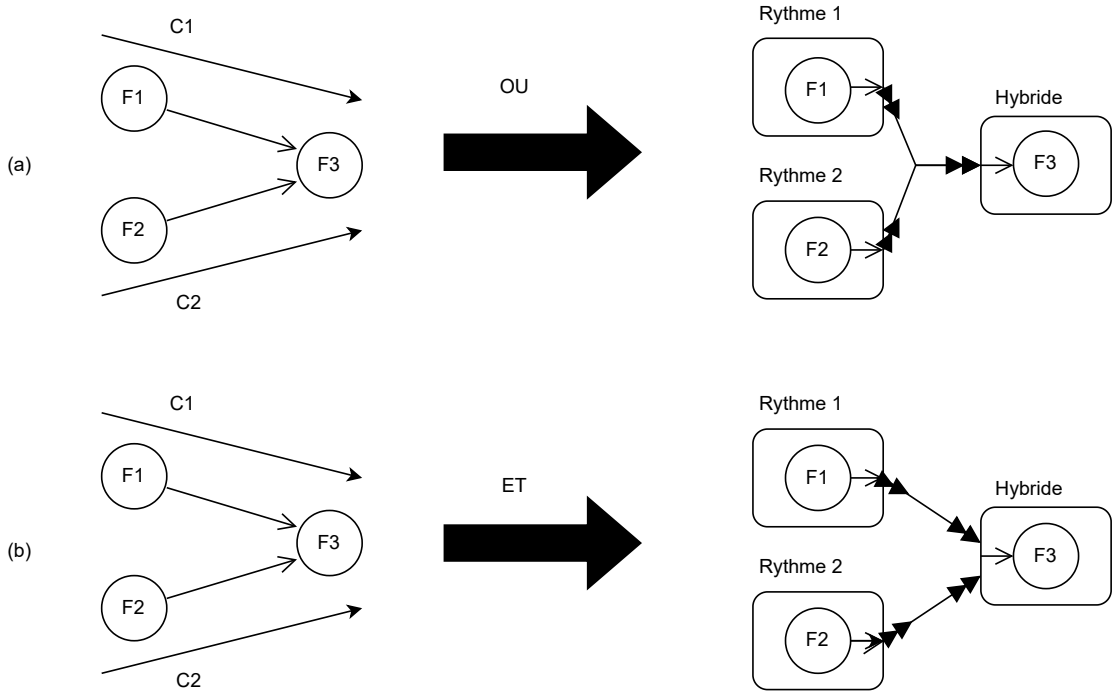

Figure 4.8: Sémantique d'activation en OU et en ET.

4.2.6 Algorithme

Nous considérons que chaque chaîne fonctionnelle est initiée par un dispositif d'entrée (capteur, bus, etc.) comme illustré sur la figure 4.3. Un dispositif d'entrée e est caractérisé par une période (en réalité délai entre deux activations ou scrutations successives) minimale T^e_{min} et une période maximale T^e_{max} pouvant être infinie. De plus, il est caractérisé par la possibilité, ou non, d'activer une interruption matérielle lorsqu'il est prêt à être lu. Nous dénommons cette possibilité *push*. L'inverse est dénommé *pull*. Ainsi, par exemple, les capteurs analogiques n'ont pas cette possibilité, alors que tout bus de communication (série, USB, Ethernet, CAN, clavier, souris, etc.) possède cette possibilité. Il faut noter que même si un capteur est de type *push*, on peut tout à fait implémenter une lecture *pull*, c'est-à-dire laisser l'interruption matérielle stocker la donnée reçue dans un buffer, et venir scruter périodiquement le buffer comme si nous avions affaire à un capteur *pull*. Cependant, l'inverse n'est pas possible sans une aide matérielle spécifique,

et sur un capteur analogique et surtout un convertisseur analogique-numérique classique utilisé pour l'acquérir, il n'est pas possible de déclencher une interruption matérielle sur valeur spécifique de la grandeur physique mesurée. Ainsi, une entrée de type *pull* ne peut être que scrutée, ce qui implique un délai supplémentaire de traitement d'une période de la tâche de scrutation à cause de l'asynchronisme entre valeur surveillée et tâche d'acquisition. Une entrée de type *push* peut déclencher une tâche de façon synchrone ou bien être scrutée comme si elle était de type *pull*. La première implémentation est synchrone, alors que la seconde est asynchrone. Nous choisirons donc, sauf lorsque cela est imposé dans le domaine (voir domaine automobile en section 4.2.8), l'implémentation synchrone, plus réactive.

De plus, nous supposons qu'un dispositif d'entrée e est lié à une et une seule fonction, qui possède uniquement ce dispositif d'entrée comme prédécesseur. Cela est consistant avec les démarches de décomposition fonctionnelle dans la plupart des méthodes de spécification fonctionnelle que nous avons présentées. Lorsque l'on arrive à une décomposition fonctionnelle de dernier niveau, nous supposons que chaque dispositif d'entrée est lu par une fonction dédiée, apparaissant en première fonction d'une chaîne fonctionnelle.

Nous considérons chaque chaîne fonctionnelle dans l'ordre d'urgence décroissant, en commençant par le dispositif d'entrée, puis en considérant chaque fonction en suivant l'ordre dans la chaîne. Cela implique notamment que toute fonction possède une fonction prédécesseur, en dehors de la première fonction d'une chaîne fonctionnelle suivant un dispositif d'entrée dans le DAG de fonctions.

Initialement, si le dispositif d'entrée e est de type *pull*, nous créons une tâche périodique de période $T^{min} = T^{max} = T_e^{min}$. Dans le cas où le dispositif est de type *push*, nous créons une tâche sporadique de période minimale $T^{min} = T_e^{min}$ et de période maximale $T^{min} = T_e^{max}$. Il faut noter que, sur un système tolérant aux fautes, il est très probable que $T_e^{max} = +\infty$. La tâche créée, qu'elle soit périodique ou sporadique, devient la tâche courante τ.

Maintenant, nous parcourons la chaîne fonctionnelle. Pour chaque fonction f apparaissant séquentiellement dans la chaîne fonctionnelle :

- Si f est déjà assignée à τ, rien à faire pour la fonction courante. Pas de surcoût au délai de bout en bout maximal de la chaîne fonctionnelle courante.
- Sinon si f est déjà assignée à une autre tâche que nous nommerons $\tau' \neq \tau$
  - Si τ' est activée de façon synchrone par boîte aux lettres, ajouter un message en OU de τ vers cette boîte aux lettres. Ce message transporte la donnée envoyée par le prédécesseur de f dans la chaîne fonctionnelle vers f. Il y a donc communication par boîte aux lettres en OU entre τ et τ'. La chaîne fonctionnelle courante paie le prix d'une communication synchrone entre τ et τ'.
  - Sinon
    - Si τ et τ' sont périodiques de même période, fusionner τ et τ'. Pas de surcoût au délai de bout en bout maximale de la chaîne fonctionnelle courante.
    - Sinon, la communication entre la fonction prédécesseur de f et f est une communication asynchrone par tableau noir, entre τ et τ'. La chaîne fonctionnelle courante paie le prix de l'asynchronisme entre τ et τ'.

- Sinon (i.e., la fonction f n'est pas affectée)
  - Si f ne possède pas de prédécesseur en dehors de la chaîne fonctionnelle courante, affecter f à τ. Pas de surcoût au délai de bout en bout maximale de la chaîne fonctionnelle courante.
  - Sinon
    * Si la période maximale de τ : $T^{max} = +\infty$ alors créer une nouvelle tâche τ', activée par boîte aux lettres par τ. Le message transmis est la donnée envoyée par le prédécesseur de f à f. La tâche courante est dorénavant τ'. La chaîne fonctionnelle courante paie le prix d'une communication synchrone entre τ et τ'.
    * Sinon affecter f à τ. Pas de surcoût au délai de bout en bout maximal de la chaîne fonctionnelle courante.

  , passer la séquence de fonctions déjà assignées dans la chaîne fonctionnelle jusqu'à atteindre une fonction non assignée. La tâche courante est celle où réside la dernière fonction assignée de la séquence. Si une telle fonction n'existe pas, l'assignation de la chaîne est terminée.

- Si la fonction n'est pas assignée et pilotée par des événements, créer une nouvelle tâche sporadique avec une période d'inter-arrivée minimale égale à la période minimale T^{min} de l'entrée. Cette tâche devient la tâche actuelle. Il n'y a pas de choix ici si nous voulons que la fonction soit déclenchée par l'événement d'entrée.

- Si la fonction source est non assignée et pilotée par le temps, nous supposons que les périodes minimale et maximale des entrées sont égales. Si une tâche périodique avec la même période existe déjà, ajouter la fonction source à cette tâche, qui devient la tâche actuelle. Ici encore, il n'y a pas beaucoup de choix. Noter que pour réduire le nombre de tâches, puisqu'une tâche représente un modèle d'activation, nous pouvons simplement utiliser une seule tâche pour chaque modèle d'activation donné.

Pour chaque fonction suivante dans la chaîne fonctionnelle :

- Si la fonction est déjà assignée à une tâche, cela signifie que la fonction fait également partie d'une chaîne de priorité supérieure, que nous avons déjà traitée.
  - Si la tâche est activée par une communication synchrone d'une autre tâche, ajouter une communication synchrone en OU entre la tâche actuelle et cette tâche, qui devient le nouvelle tâche active. Comme la tâche hébergeant la fonction est déjà déclenchée par une communication synchrone, ajouter une communication synchrone entre la tâche actuelle et la tâche exécutant la fonction est le choix qui minimise la pénalité de bout en bout ajoutée.
  - Sinon, si la tâche est activée par un événement externe ou est périodique, ajouter une communication asynchrone entre la tâche actuelle et cette tâche. Dans ce cas, nous n'avons effectivement pas le choix : puisque la tâche exécutant la fonction a déjà sa propre source de déclenchement, nous ne pouvons pas en ajouter une autre. Par conséquent, nous devons payer le coût d'une communication asynchrone.

    Sinon (la fonction n'est pas assignée à une tâche),

- Si la tâche actuelle est périodique, attribuer la fonction à la tâche actuelle. La chaîne fonctionnelle actuelle est la chaîne la plus prioritaire utilisant la fonction, et l'insertion de la fonction dans la tâche actuelle réduira donc le délai de bout en bout par rapport à tout autre choix.
- Sinon (la tâche actuelle est sporadique),
  - Si la fonction apparaît dans une autre chaîne fonctionnelle de priorité inférieure, créer une nouvelle tâche et attribuer la fonction à cette tâche. Créer une communication synchrone entre la tâche actuelle et cette nouvelle tâche, qui devient la tâche actuelle. Nous devons faire ce choix, qui n'est pas aussi bon pour le délai de bout en bout de la chaîne actuelle que si nous avions inséré la fonction dans la tâche actuelle. Néanmoins, nous ne pouvons pas le faire parce que la tâche actuelle est sporadique et activée par un événement. Si l'événement ne se produit pas (période maximale infinie), alors si la chaîne de priorité inférieure utilisant la fonction devait être exécutée, la fonction ne serait pas exécutée et arrêterait la chaîne fonctionnelle de priorité inférieure.
  - Sinon (la fonction n'apparaît dans aucune autre chaîne fonctionnelle), attribuer simplement la fonction à la tâche actuelle.
    L'algorithme 2 résume les étapes et les règles détaillées de la méthode rythmes ci-dessus.

4.2.7 Discussion sur l'optimalité

Lignes 2, 3, 7 : La première fonction de la chaîne, que nous appellerons $F1$, si elle est de type pull, est affectée dans une tâche périodique de période T^{min} qui scrute la valeur d'entrée (lecture asynchrone). À part réduire la période de scrutation qui aurait pour effet de sur-échantillonner la valeur lue en augmentant la charge processeur, il n'y a pas de moyen de réduire le délai de bout en bout entre franchissement d'un seuil par la valeur scrutée et terminaison de la fonction f_1. Ce délai maximal d_1 est la somme de la période de la tâche et du pire temps de réponse de la tâche qui selon nos hypothèses ne dépassera pas la période de la tâche. Par conséquent $d_1 = 2 \times T^{min}$.

Lignes 4, 5, 7 : Si $F1$ est de type push, la tâche créée est sporadique de période minimale T^{min}, déclenchée de façon synchrone par l'événement push. Ce type de déclenchement minimise le délai de bout en bout entre arrivée d'un événement push et fin d'exécution de la fonction $F1$, qui est, sous nos hypothèses, $d_1 = T^{min}$. Notons que si on avait décidé de scruter cette entrée avec une tâche périodique de période T^{min}, le délai aurait été $2 \times T^{min}$.

À partir de ce rythme initial de tâche, nous parcourons (à partir de la ligne 8), les fonctions successives de la chaîne fonctionnelle. Notons que lorsque nous considérons la chaîne la plus importante, aucune fonction de la chaîne n'a été assignée. Par conséquent, si la période T^{max} de la tâche courante n'est pas infinie, toutes les fonctions sont fusionnées dans la même tâche. Le délai de bout en bout maximal suivant nos hypothèses est donc le même délai que d_1. Si toutefois T^{max} est infinie, alors à chaque fois que nous rencontrons une fonction ayant des prédécesseurs faisant partie d'une autre chaîne fonctionnelle (join) comme sur le haut de la figure 4.7, nous sommes contraints, afin d'éviter qu'un autre délai

Algorithme 2 Allocation des fonctions aux tâches

Pour chaque chaîne fonctionnelle cf dans l'ordre d'urgence décroissante **Faire**
  Si le dispositif d'entrée de cf est de type *pull* **Alors**
    Créer une nouvelle tâche périodique τ pour cf avec une période T_τ appropriée, lisant le dispositif d'entrée de façon asynchrone.
  Sinon
    Créer une nouvelle tâche sporadique τ pour cf avec une période minimale T_τ appropriée, activée par le dispositif d'entrée de façon synchrone.
  Fin Si
  Placer la première fonction de cf dans τ.
  Pour chaque fonction f dans l'ordre de la chaîne fonctionnelle cf **Faire**
    Si f est déjà assignée à τ **Alors**
      Ne rien faire pour la fonction courante.
    Sinon Si f est déjà assignée à une autre tâche $\tau' \neq \tau$ **Alors**
      Si τ' est activée de façon synchrone par boîte aux lettres **Alors**
        Ajouter un message en OU de τ vers la boîte aux lettres de τ'.
        Il y a communication par boîte aux lettres en OU entre τ et τ'.
        La chaîne fonctionnelle courante paie le prix d'une communication synchrone entre τ et τ'.
      Sinon
        Si τ et τ' sont périodiques de même période **Alors**
          Fusionner τ et τ'.
        Sinon
          La communication entre la fonction prédécesseur de f et f est une communication asynchrone par tableau noir, entre τ et τ'.
          La chaîne fonctionnelle courante paie le prix de l'asynchronisme entre τ et τ'.
        Fin Si
      Fin Si
    Sinon
      Si f ne possède pas de prédécesseur en dehors de la chaîne courante **Alors**
        Affecter f à τ.
      Sinon
        Si La période maximale de τ est $+\infty$ **Alors**
          Créer une nouvelle tâche τ', activée par boîte aux lettres par τ.
          Le message transmis est la donnée envoyée par le prédécesseur de f à f.
          La tâche courante devient τ'.
          La chaîne courante paie le prix d'une communication synchrone entre τ et τ'.
        Sinon
          Affecter f à τ.
        Fin Si
      Fin Si
    Fin Si
  Fin Pour
Fin Pour

maximal de bout en bout soit infini, de faire le choix (c) de la figure 4.7 qui minimise le délai sur la chaîne en cours, en garantissant qu'aucun autre délai maximal ne soit infini.

Si la chaîne considérée n'est pas d'urgence maximale, alors on peut rencontrer des fonctions déjà présentes dans des tâches faisant partie de chaînes plus urgentes que la chaîne fonctionnelle en cours. Supposons que la chaîne considérée est $C2$ par rapport à la figure 4.7, que nous venons de considérer $F2$, et que nous considérons maintenant $F3$, qui fait partie de la chaîne $C1$ plus prioritaire, et qui par conséquent a déjà été assignée à une tâche. Notons qu'il y a deux cas tels qu'énoncés dans le paragraphe précédent : soit la fonction se trouve dans une tâche activée par une boîte aux lettres (cas (c) de la figure 4.7), que l'on peut aussi activer par boîte aux lettres à partir de la tâche contenant la fonction $F2$, ce qui impliquera un surcoût de délai de bout en bout de la période minimale de la tâche hybride contenant $F3$; soit la fonction se trouve dans une tâche ayant son propre rythme d'activation (choix (a) de la figure 4.7), auquel cas la communication entre la tâche exécutant $F2$ et la tâche exécutant $F3$ ne peut être qu'asynchrone. Cela entraîne un surcoût sur le délai de bout en bout de la période maximale (asynchronisme) plus la période minimale (hypothèse de temps de réponse maximal) de la tâche exécutant f_3 sur le délai maximal entre fin de $F2$ et fin de $F3$.

Par construction, les pires délais de bout en bout sont minimums pour la chaîne la plus prioritaire, en garantissant que le délai sur d'autres chaînes ne peut être infini, et d'une façon générale pour une chaîne fonctionnelle de niveau d'urgence donné, RYM est tel que le pire délai de bout en bout est minimisé avec respect du fait que les chaînes plus urgentes ont vu leur délai minimisé tout en garantissant qu'aucune chaîne moins urgente ne peut voir son délai maximal être infini.

Nous pourrions caractériser cette affectation comme étant optimale, avec un sens local, comme Rate Monotonic minimise le temps de réponse d'une tâche de priorité donnée, après avoir minimisé le temps de réponse de toutes les tâches plus prioritaires. Cependant, d'une manière générale, au regard du problème d'ordonnançabilité, RM n'est pas optimal, contrairement à EDF. Il en va de même pour RYM qui n'est pas optimal pour le respect des échéances données aux chaînes fonctionnelles, mais attribut une affectation de tâches aux fonctions qui est optimale au sens de minimiser un délai maximal de bout en bout pour une chaîne de priorité donnée, en ayant d'une part minimisé le délai pour les chaînes plus prioritaires, et d'autre part garanti l'absence de délais infinis.

4.2.8 Cas particuliers

Il peut arriver que l'entrée d'une chaîne fonctionnelle soit un chien de garde (watchdog), qui est en général activé suite à la non-arrivée d'un événement sporadique. Le watchdog lui-même est représentable par le rythme *Timed* en AADL, cela correspond, d'un point de vue comportement temporel, à une tâche sporadique ayant comme période minimale le délai de garde (par exemple, watchdog activé si aucun message n'a été reçu depuis 100 ms). La période maximale est l'infini, ce qui correspond au cas où l'événement surveillé arrive toujours dans le délai de garde. Un watchdog peut donc être représenté comme un dispositif de type *push* ayant comme période minimale le délai du watchdog, et comme période maximale l'infini.

Il arrive que plusieurs capteurs, en particulier analogiques, doivent être scrutés à la même période. Dans ce cas, il est intéressant de fusionner leurs fonctions d'acquisition dans la même tâche. Cela est fait par notre algorithme.

Dans certains secteurs industriels, comme dans l'automobile, même lorsqu'il est techniquement possible d'activer une tâche sur l'arrivée d'un événement (typiquement, un message arrivant sur un réseau CAN), il arrive que la scrutation soit préférée par les concepteurs. Cela est moins réactif (c.-à-d. le délai de bout en bout entre l'arrivée d'un message et la chaîne fonctionnelle liée est plus long) qu'une conception sporadique. En effet, on paie en périodique une période de plus que dans le cas sporadique dans le délai de bout en bout. Cependant, le fait de maîtriser toute la chaîne, et de faire de l'*offset free* sur les tâches permet une plus grande ordonnançabilité que lorsqu'un instant critique peut exister. Dans ce cas, le bus CAN doit être considéré comme un capteur de type *pull*. Afin de forcer notre algorithme à scruter un dispositif d'entrée, il suffit de déclarer celui-ci comme étant de type *pull* et de lui affecter comme période la période de scrutation voulue.

4.3 Étude de cas

4.3.1 Application

L'exemple concerne un système de contrôle de mine, qui est illustré sur la figure 4.9.

Le système doit surveiller le niveau d'eau dans une mine d'extraction qui est inondée naturellement par infiltration d'eau, afin que les machines d'extraction puissent y travailler. Le niveau d'eau doit être maintenu entre un niveau bas et un niveau élevé pour faciliter le travail des excavatrices, et à cette fin, une pompe d'extraction d'eau est utilisée.

En plus de contrôler le niveau d'eau, le système doit également contrôler le niveau de méthane, car il est possible qu'une poche gazeuse soit percée lors de l'extraction des minerais. En cas de dépassement du seuil d'alerte de méthane, appelé niveau-1, le système doit évacuer les employés de la mine. En cas de dépassement du second seuil d'alerte, appelé niveau-2 (niveau-1 $<$ niveau-2 du point de vue sécurité), outre l'alerte donnée aux employés de la mine, le système doit également désactiver la pompe d'extraction d'eau pour éviter un incendie.

Le système doit réagir (c.-à-d. couper la pompe et/ou allumer l'alarme) en moins de *200 ms* lorsqu'un seuil d'alarme franchi.

Dans cet exemple, nous supposons que tous les capteurs sont passifs (i.e., de type *pull*), c'est-à-dire que les données sont scrutées périodiquement. La scrutation du capteur de méthane pourrait se faire à une période de *50 ms*. De plus, le contrôle de niveau d'eau peut être fixé initialement à *500 ms*. Ces périodes choisies par le concepteur sont indicatives, et si les délais de bout en bout ne respectent pas les échéances, elles pourront être réduites. De même, si les pires délais de bout en bout sont sensiblement inférieurs aux échéances de bout en bout, ces périodes peuvent être augmentées.

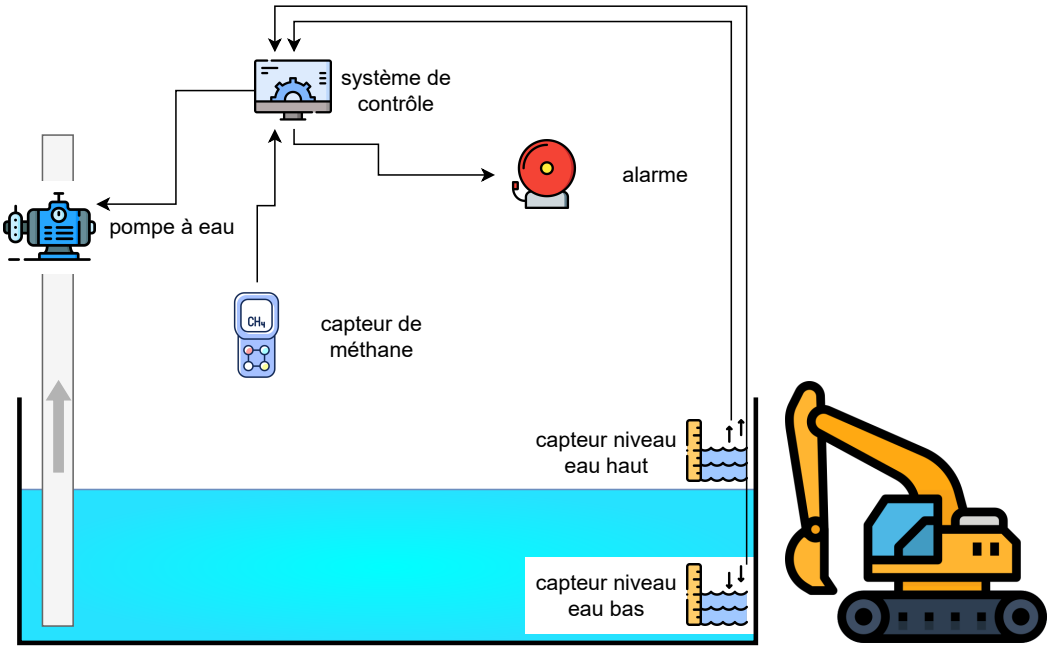

Figure 4.9: Illustration du système de contrôle de la mine

Le diagramme de structure fonctionnelle donné sur la figure 4.10 illustre les interconnexions entre les différentes fonctions et les chaînes fonctionnelles du système, ainsi que les contraintes de temps associées. Ainsi, par exemple, la chaîne fonctionnelle C3 est constituée de l'enchaînement de trois fonctions, allant de la lecture de niveau de méthane à la commande de l'alarme, dont l'échéance de bout en bout est de 200 ms. Notons que puisque les capteurs de niveau d'eau sont deux capteurs tout ou rien qui doivent être lus tous les deux pour connaître le niveau d'eau, on peut exprimer cet ensemble de capteurs comme un seul.

Chaque chaîne fonctionnelle cf_i est caractérisée par un délai de bout en bout, qui représente le temps maximal autorisé pour qu'un élément de données circule à travers le flot de données. Une priorité plus élevée est accordée aux chaînes fonctionnelles d'urgence relative plus élevée.

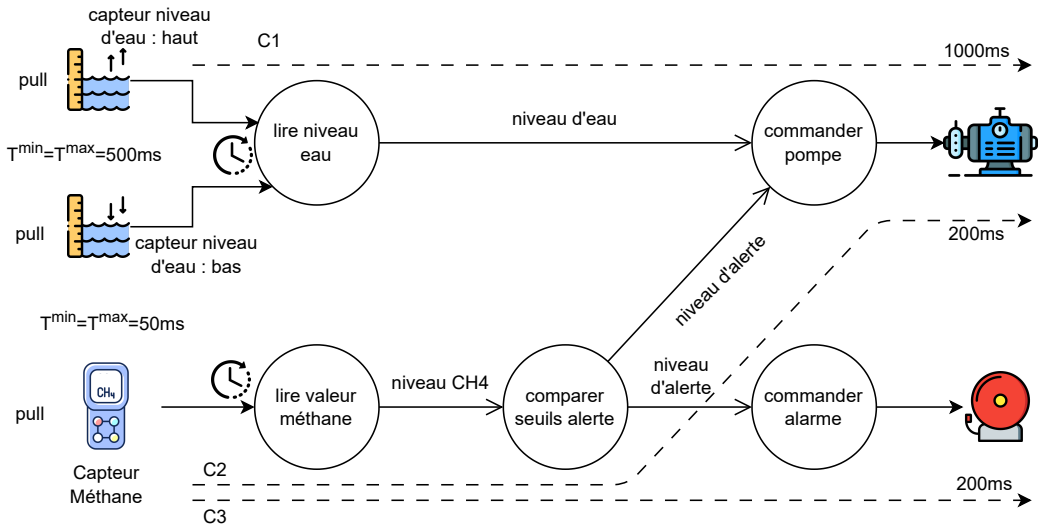

Figure 4.10: Modèle fonctionnel du système de contrôle de la mine

En appliquant la méthode des rythmes à l'exemple sur la chaîne C3 à partir du dispositif d'entrée *Capteur méthane*, en observant que le dispositif d'entrée est de type *pull*, nous créons une tâche périodique avec une période de 50 ms (voir figure 4.11), dans laquelle nous plaçons la fonction *lire valeur méthane*, cela correspond aux lignes 1 à 5 de l'algorithme 2. *Comparer seuils alerte*, qui ne possède qu'un prédécesseur, est donc placée dans la même tâche, et il en va de même pour *Commander alarme* (lignes 27 à 29).

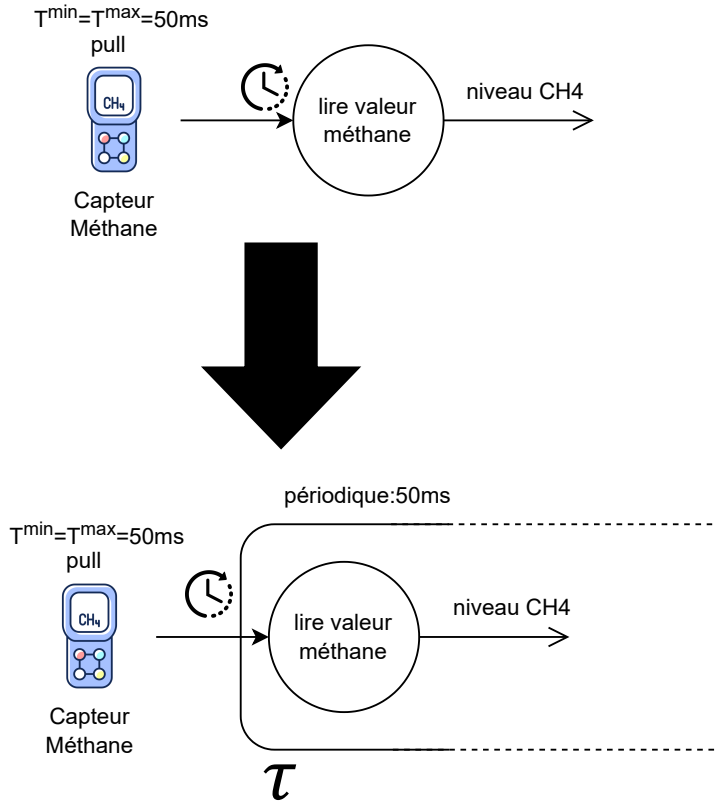

Figure 4.11: Création de tâche pour acquisition de données à partir du capteur de méthane

Pour la chaîne C2, nous avançons sans changement sur les fonctions *lire valeur méthane* et *Comparer seuils alerte* qui sont déjà placées dans une tâche. Puis, nous arrivons sur *commander pompe* qui ne possède qu'un prédécesseur, et est donc placée dans la même tâche (lignes 27 à 29).

Le pire délai de bout en bout du dépassement de seuil d'alerte de méthane à la commande de pompe et la commande d'alarme est donc de deux fois (asynchronisme dû au capteur de type *pull*) la période de la tâche, soit 100 ms. Notons que c'est largement inférieur aux 200 ms demandées. On pourrait donc choisir de monter la période de la tâche jusqu'à 100 ms, divisant par deux la charge processeur créée par les fonctions de ces chaînes fonctionnelles. Le concepteur peut souhaiter s'assurer un peu de marge à cause des délais technologiques : durée de la lecture d'un niveau de méthane, de l'ordre de quelques micro secondes (conversion analogique-numérique et propagation électrique), commande analogique de la pompe pouvant prendre plusieurs millisecondes, durée de la communication série basse vitesse avec l'alarme, pouvant aussi prendre quelques millisecondes.

Concernant la chaîne C1, comme les capteurs d'eau sont de type tout ou rien, de nombreux dispositifs d'acquisition de ce type de signal sont capables de déclencher des interruptions matérielles lors d'un changement. Ils sont donc de type *push*. Cependant, il est difficile d'établir un délai minimal entre deux changements d'état successifs de ces capteurs (on peut imaginer un clapotis de l'eau faisant changer l'état d'un capteur plusieurs milliers de fois par seconde). Utiliser un déclenchement apériodique, non contrôlé, d'une tâche sur un dispositif de ce type risque de poser des difficultés lors de la phase d'étude d'ordonnançabilité du système. De plus, nous savons que cette chaîne fonctionnelle n'a pas besoin d'être très réactive puisque le délai de bout en bout toléré est de l'ordre de la seconde. C'est typiquement le genre de situation où, bien que les capteurs autorisent du *push*, on préférera, par mesure de simplicité, sacrifier à la réactivité du système en venant scruter périodiquement les capteurs, c'est-à-dire en les utilisant en mode *pull*. C'est la raison pour laquelle le concepteur a choisi d'exprimer le fait qu'on vienne scruter les capteurs à une période de 500 ms. Nous créons donc une tâche périodique à cette période, dans laquelle la fonction *lire niveau eau* est placée (lignes 3 et 4). Puisque la fonction *commande pompe* est déjà mappée dans une tâche, nous devons opérer entre les deux tâches une communication asynchrone, payant l'asynchronisme des deux tâches dans le délai de bout en bout (lignes 12, 17, 21 à 23). Le pire délai obtenu tient compte de deux lectures asynchrones : celle du niveau d'eau par la tâche de période 500 ms, puis l'asynchronisme entre cette tâche et la tâche hébergeant *commander pompe*, de période 50 ms. Cela donne un pire délai de bout en bout de 1,1 seconde, ce qui est supérieur à l'échéance d'une seconde. Par conséquent, pour réduire le pire délai, on pourra, par exemple, réduire à 450 ms la période de la tâche exécutant *lire niveau eau*.

En appliquant toutes les règles présentées précédemment à l'exemple en cours, nous obtenons l'architecture multitâche illustrée dans la figure 4.12. Elle se compose de deux tâches périodiques. La première a une période de 450 ms, tandis que la seconde a une période de 50 ms. La communication de données entre les deux tâches est asynchrone.

Tableau 4.1: Caractéristiques des tâches

Tâche	WCET	priorité	Période	Échéance	Durée de partage de ressource
Fixed priority					
τ_1	170 ms	1 (basse)	450 ms	450 ms	10 ms
τ_2	30 ms	2 (élevée)	50 ms	50 ms	10 ms
EDF					
τ_1	170 ms	dynamique	450 ms	450 ms	10 ms
τ_2	30 ms	dynamique	50 ms	50 ms	10 ms

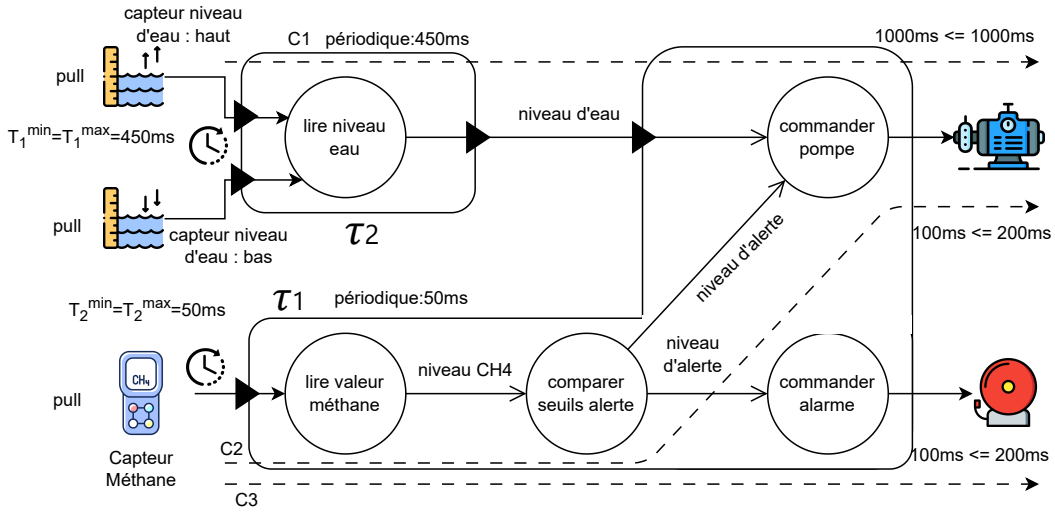

Figure 4.12: Modèle de pré-déploiement du système de contrôle de la mine

4.3.2 Étude d'ordonnançabilité

Nous supposons que nous nous trouvons quelques jours, semaines, ou mois plus tard, en fonction de la taille et des efforts de développement requis. Désormais, une implémentation sur la cible nous permet de connaître les WCET des tâches.

Nous supposons que le système d'exploitation sous-jacent propose plusieurs politiques d'ordonnancement, comme Linux. Dans ce qui suit, nous effectuerons une analyse d'ordonnancement avec les politiques d'ordonnancement à priorités fixes aux tâches (FP) et à échéance la plus proche (EDF). Comme protocole de partage des ressources, nous choisirons le protocole de plafonnement des priorités immédiat, offert dans POSIX pthreads. Le Tableau 4.1 montre les caractéristiques des tâches de conception, τ_1 et τ_2, avec des WCET respectifs de 170 ms et 30 ms. Elles partagent une variable (le tableau noir de communication), dont les accès sont protégés pour garantir l'exclusion mutuelle. Nous supposons que la durée maximale qu'une tâche peut passer à l'intérieur de la section critique partagée est de 10 ms.

MAST[MAS] est l'outil que nous avons utilisé pour tester si le système est ordonnançable ou non. La technique d'analyse d'ordonnancement repose sur le critère du temps

de réponse. Le système de tâches est ordonnançable et chaque tâche se termine dans les délais impartis selon les deux politiques d'ordonnancement, FP (voir figure 4.13) et EDF (voir figure 4.14).

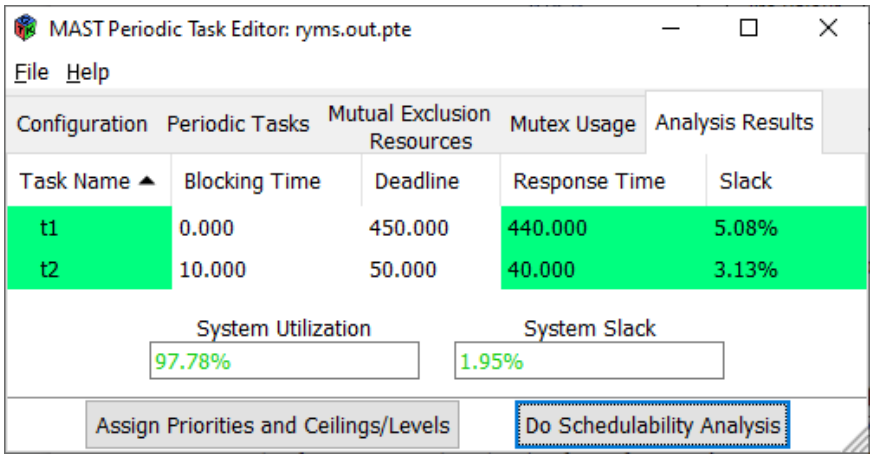

Figure 4.13: Résultat de l'ordonnancement selon la politique RM

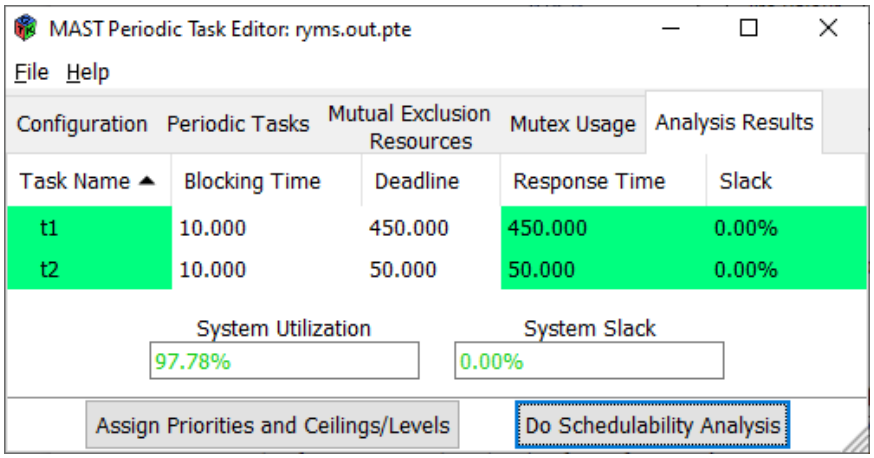

Figure 4.14: Résultat de l'ordonnancement selon la politique EDF

4.3.3 Extension du système de contrôle de la mine

Afin de montrer que RYM va au-delà d'un système très simple tel que celui que nous avons présenté en section précédente, nous complexifions le système afin de mixer les différents types de rythme d'activation. Le nouveau système vise à intégrer des aspects tolérance aux fautes, montrant l'importance notamment de tenir compte de la période maximale de certains rythmes, pouvant être infinie, par exemple en cas de panne. Le système étendu compte désormais 5 chaînes fonctionnelles ($C1$ à $C5$) contre 3 initialement comme le montre la figure 4.15.

Les caractéristiques essentielles des dispositifs d'entrée (nature, type et périodicité) sont résumées dans le tableau 4.2, tandis que les informations sur les chaînes fonctionnelles (source, destination et l'échéance) sont synthétisées dans le tableau 4.3. Ce qui change ici est que le capteur de méthane est un capteur USB envoyant, normalement, des trames contenant le niveau de méthane mesuré à une fréquence 20 hertz.

La fonction *lire méthane* a été divisée en 3 fonctions :

- **accumuler octets trame :** collecte les octets de la trame méthane en sortie.
- **vérifier checksum :** vérifie l'intégrité de la trame méthane et ne produit en sortie une trame que si celle-ci est correcte. Dans le cas contraire, la chaîne fonctionnelle $C3$ s'arrête. Un booléen est envoyé vers Filtrer fautes pour signaler le succès ou non du calcul de checksum.
- **extraire niveau méthane :** extrait la valeur du niveau de méthane à partir de la trame (si elle arrive, i.e., si le checksum était correct).

L'ajout du mécanisme de détection de panne par watchdog et les chaînes $C4$ et $C5$ permettent d'approfondir l'étude de la tolérance aux fautes avec la méthode des rythmes. Les nouvelles sous-fonctions de $C3$ donnent une granularité plus fine du système pour étudier différents mappings.

La fonction *filtrer fautes* représente un mécanisme généralement appelé FDIR (*Fault Detection, Isolation&Recovery*), son rôle ici est de faire le lien entre faute et panne de capteur. Sans détailler son fonctionnement interne, il reçoit en entrée les succès et échecs de vérification de checksum, ainsi que les événements timeout générés par le chien de garde lorsqu'aucune trame n'a été reçue depuis 100 ms. Il implémente une stratégie de choix de déclaration de panne ou de fonctionnement acceptable du capteur USB. Il fait partie de quatre chaînes fonctionnelles, les deux initiées par l'arrivée d'une trame, et les deux initiées par le déclenchement du chien de garde. La fonction *filtrer fautes* dispose de deux options pour décider de la présence d'une panne nécessitant le déclenchement d'une alerte. La première approche consiste à considérer chaque faute détectée par *vérifier checksum* ou chaque retard de plus de 100 ms de manière individuelle. Ainsi, la présence d'une seule faute suffit à indiquer une défaillance et à activer immédiatement une alerte, permettant de respecter l'échéance temps-réel contraignante de 200 ms. Toutefois, cette réaction *faute par faute* ne permet pas de distinguer des fautes transitoires ou intermittentes de véritables pannes permanentes. La seconde approche cherche à pallier cet inconvénient en cumulant les fautes sur une fenêtre glissante des k dernières mesures avant de prendre une décision. Ce filtrage rend la détection plus robuste aux perturbations transitoires. Cependant, le délai nécessaire au cumul des fautes empêche le respect de l'échéance de 200 ms. Compte tenu des contraintes temps-réel, seule la première solution "faute par faute" semble viable pour cet exemple, malgré sa sensibilité accrue aux fautes non persistantes. L'implémentation de la fonction *filtrer fautes* constitue donc un choix de conception entre rapidité de réaction et robustesse de détection.

Dispositif	**Nature**	**Type**	**Période min**	**Période max**
Capteur niveau d'eau	Numérique	Pull	$T^{min} = 450ms$	$T^{max} = 450ms$
Capteur méthane	Port USB	Push	$T^{min} = 50ms$	$T^{max} = +\infty$
Chien de garde	Numérique	Push	$T^{min} = 100ms$	$T^{max} = +\infty$

Tableau 4.2: Caractéristiques des dispositifs d'entrée

Chaîne	Source	Destination	Échéance
C1	Capteur niveau d'eau	Pompe	1 s
C2	Capteur méthane	Alarme	200 ms
C3	Capteur méthane	Pompe	200 ms
C4	Chien de garde	Pompe	200 ms
C5	Chien de garde	Alarme	200 ms

Tableau 4.3: Résumé des chaînes fonctionnelles

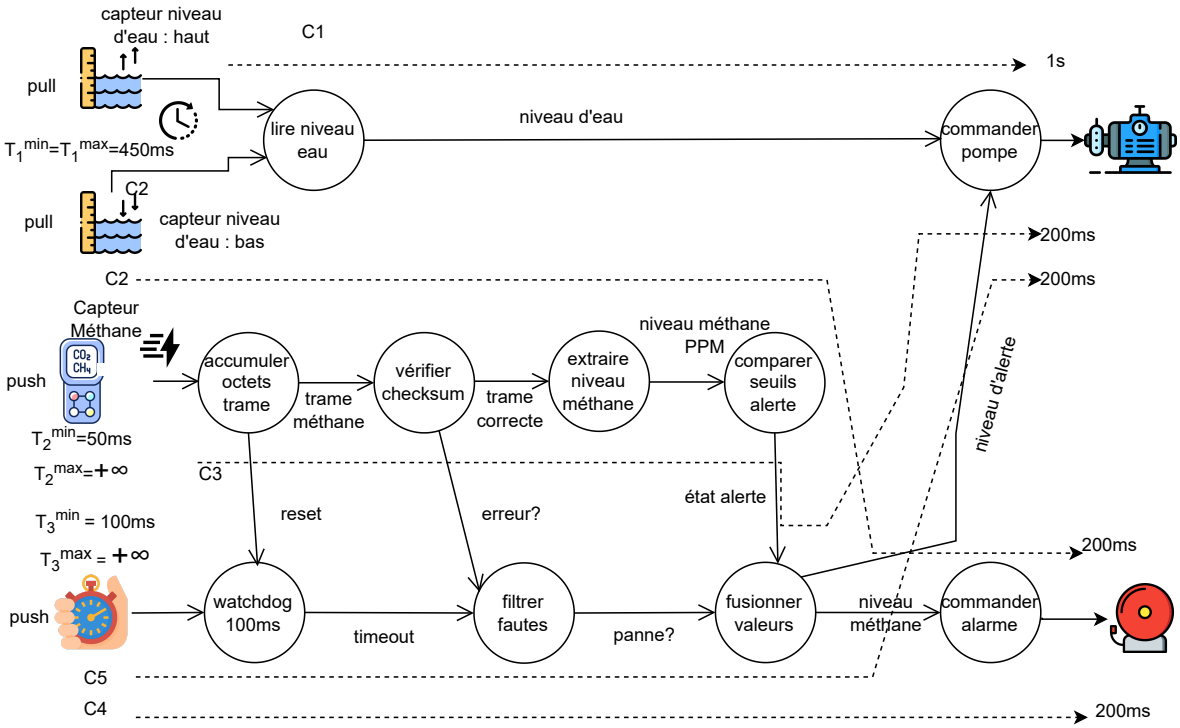

Figure 4.15: Modèle fonctionnel du système étendu de contrôle de la mine

L'application de la méthode des rythmes a conduit à une architecture composée de 4 tâches comme le montre la figure 4.16 :

- **la tâche τ_1 :** périodique de période 450 ms, exécute la fonction *lire niveau eau* associée à la chaîne fonctionnelle C1.
- **la tâche τ_2 :** sporadique, de période minimale 50 ms, exécute les fonctions responsables de la récupération et du traitement des trames provenant du capteur USB de méthane.
- **la tâche τ_3 :** exécute la fonction watchdog selon un rythme *Timed* de 100 ms. Elle est réinitialisée par la tâche τ_2 via une communication de type *reset* (événement) lorsque cette dernière est activée.
- **la tâche τ_4 :** de nature hybride, est déclenchée par les tâches τ_2 et τ_3 en OU à travers une boîte aux lettres. Elle reçoit de la tâche τ_2 par communication asynchrone l'information d'état d'alerte. Notons cependant que cette communication n'est pas réellement asynchrone, puisqu'une implémentation de τ_2 attendra de mettre à jour

le tableau noir de communication *comparer seuils alerte* et *fusionner valeurs* avant de placer le message dans la boîte aux lettres entre *vérifier checksum* et *filtrer fautes*. Cela implique qu'une valeur fraîche est présente dans le tableau noir à chaque fois que τ_4 est activée par τ_2. Elle reçoit également de la tâche τ_1 par communication asynchrone la donnée de niveau d'eau. La communication asynchrone entre les tâches τ_2 et τ_4 s'effectue avant le déclenchement synchrone de τ_4 par la tâche τ_2. Si aucune panne n'arrive sur le flot de données C2, la valeur est écrite dans le tableau, n'impactant pas le délai bout en bout des chaînes C2 et C3.

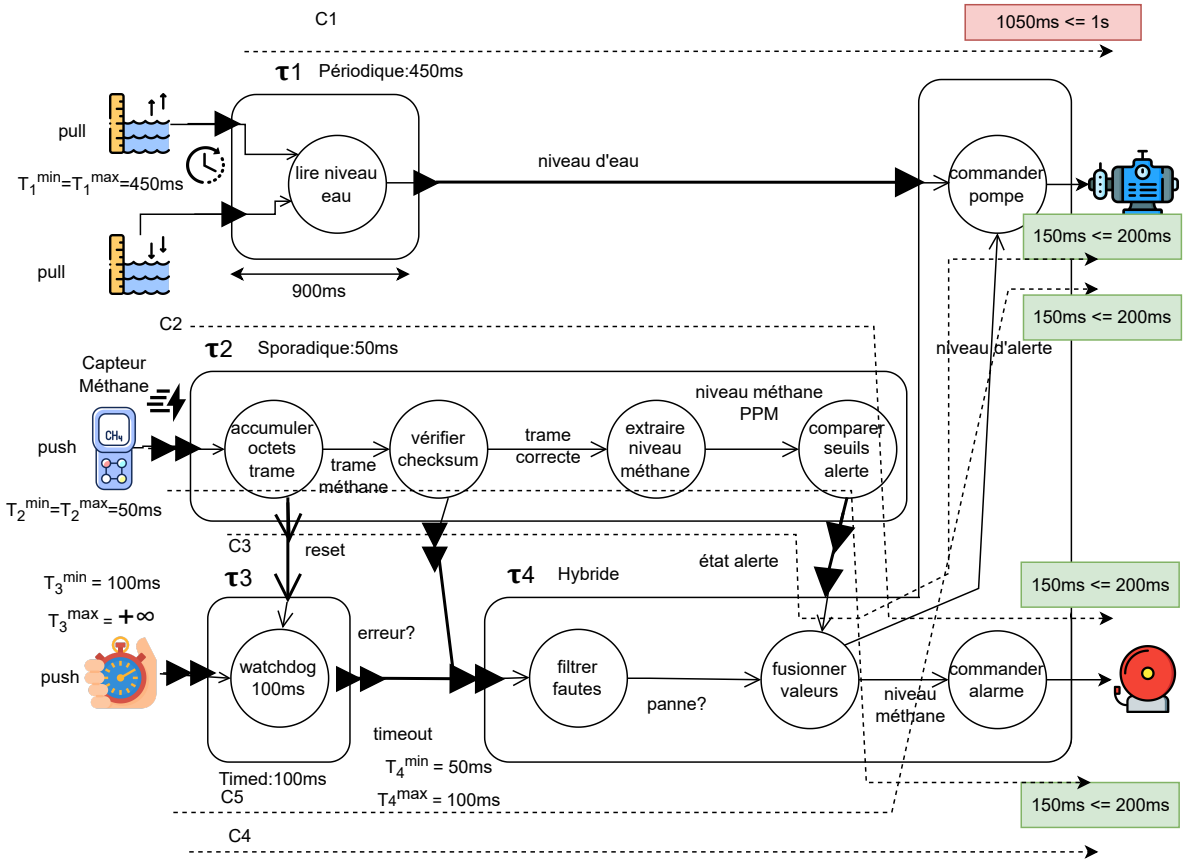

Figure 4.16: Modèle de pré-déploiement du système étendu de contrôle de la mine

La tâche hybride τ_4 a un statut particulier. En effet, elle est déclenchée suite à l'arrivée d'une trame, et entre deux de ces déclenchements, il existe un délai minimal de 50 ms, cependant, elle peut aussi être déclenchée par la non-arrivée d'une trame pendant 100 ms. Nous supposerons donc qu'elle est contrainte de s'exécuter dans un délai maximal de 50 ms (le plus petit des deux). Cependant, sa période minimale est 50 ms alors que sa période maximale est 100 ms. Le fait de déclencher en OU une tâche hybride par un événement OU l'absence de cet événement nous a permis de changer le $T^{max} = +\infty$ des tâches amont en une valeur bornée pour la tâche hybride.

Le délai de bout en bout sur la chaîne fonctionnelle C1 est calculé par la formule suivante :

$$T_1^{\max} + T_1^{\min} + T_4^{\max} + T_4^{\min} = 450 \text{ ms} + 450 \text{ ms} + 100 \text{ ms} + 50 \text{ ms} = 1050 \text{ ms}$$

dans ce cas, on remarque que le délai de bout en bout de C1 dépasse la borne de 1000 ms. On pourrait réduire la période T1 à 420 ms pour que l'échéance soit respectée et obtenir un délai de bout en bout de 990 ms qui est inférieur à l'échéance de 1000 ms.

En ce qui concerne les chaînes urgentes C2 et C3 qui ont comme échéance 200 ms, le délai de bout en bout est de 150 ms, calculé selon la formule suivante :

$$50 \text{ ms (capteur méthane)} + 50 \text{ ms } (T_2^{\min}) + 50 \text{ ms } (T_4^{\min}).$$

Concernant les chaînes C4 et C5, on a un délai de 100 ms lié à l'arrêt de réception des trames, plus T_4^{min} qui est égal à 50 ms. Le délai de bout en bout dans ce cas est égal à 150 ms.

Bien que cette solution soit satisfaisante, il est possible de l'améliorer davantage afin d'optimiser le délai de bout en bout dans le pire des cas sur la chaîne C1. Ceci est permis grâce à la nature particulière de la tâche τ_4, une tâche hybride activée par les deux tâches en amont τ_2 et τ_3. Dans ce cas précis, il est envisageable d'établir une communication synchrone en OU comme le montre la figure 4.17, contrairement aux situations où cela serait prohibé, car la tâche possède déjà son propre rythme, comme c'est le cas pour une tâche périodique. Mais, pour une tâche hybride, cela est tout à fait réalisable et va nous permettre de gagner le coût lié à l'asynchronisme (l'ancien $T_4^{\max}$ de 100ms). Ainsi, $T_4^{\max}$ devient maintenant $T_1^{\min}$ de 450ms et le délai de bout en bout devient dans ce cas :

$$T_1^{\max} + T_1^{\min} + T_4^{\min} = 450 \text{ ms } + 450 \text{ ms } + 50 \text{ ms } = 950 \text{ ms}$$

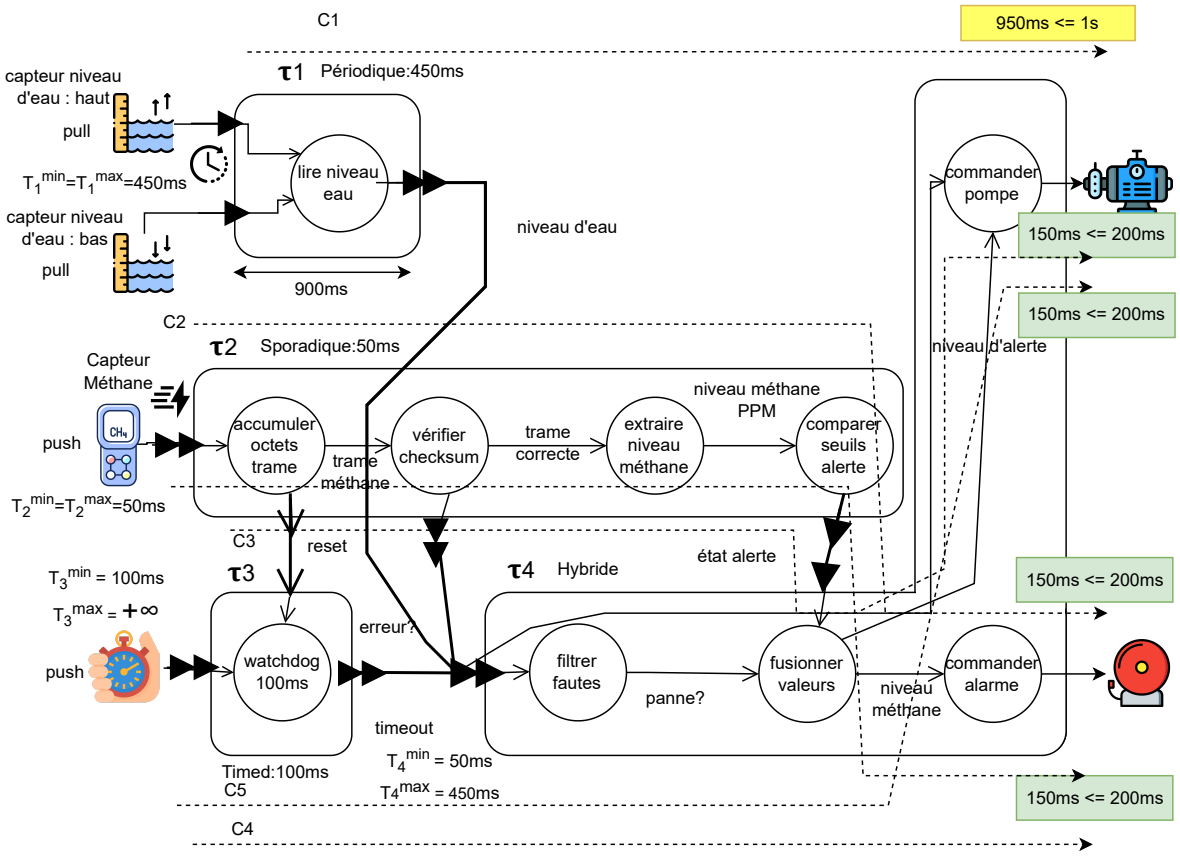

Figure 4.17: Modèle de pré-déploiement optimal du système étendu de contrôle de la mine

4.4 Comparaison

Dans cette section, nous proposons une comparaison entre les algorithmes *LA* et *JLA* présentés [BLD05] et notre méthode RYM. Ce choix de méthode vient du fait que les hypothèses sur ce qui est connu du concepteur au moment du mapping dans *LA* et *JLA* est ce que la littérature propose de plus approchant de nos hypothèses.

Cette comparaison est appliquée sur l'exemple de système temps-réel présenté dans [BLD05]. Nous appliquons les algorithmes *LA* et *JLA* sur cet exemple, puis nous utilisons notre méthode RYM pour faire la même opération de mapping.

4.4.1 Description des deux algorithmes

L'algorithme Late Activation (*LA*) analyse le DAG de fonctions et crée les tâches en deux étapes. Dans la première étape, il parcourt le graphe et crée une nouvelle tâche à chaque fois qu'une fonction possède plusieurs successeurs. Il assigne chaque fonction à une seule tâche. Si deux fonctions connectées appartiennent aux mêmes chaînes fonctionnelles, elles sont assignées à la même tâche. Cela continue jusqu'à ce qu'aucune autre fonction connectée ne puisse être fusionnée.

Dans la seconde étape, *LA* assigne les paramètres temps réel à chaque tâche en fonction des échéances des chaînes fonctionnelles. Chaque activation d'une tâche reçoit une échéance absolue égale à l'échéance minimale de toutes les chaînes fonctionnelles sortantes de cette activation.

L'algorithme Joined Late Activation (*JLA*) essaie de réduire le nombre de tâches générées par rapport à *LA*. Il est similaire à *LA* en créant d'abord des tâches à partir du DAG. La différence clé est que lors de l'assignation d'une fonction à une tâche, il examine d'abord les successeurs sur la chaîne fonctionnelle à échéance minimale. Si un tel successeur n'a qu'un seul lien entrant, il est fusionné dans la même tâche. Cette approche tend à joindre plus de fonctions dans les tâches par rapport à *LA*.

Les relations de précédence dans le DAG deviennent des relations de précédence entre les tâches générées dans les deux algorithmes. Les tâches sont ordonnancées par EDF en fonction de leurs échéances assignées.

Une différence clé entre *LA* et *JLA* est que dans *LA*, une tâche est créée pour chaque branche de successeurs multiples. Dans *JLA*, les successeurs sur les chaînes fonctionnelles à échéance minimale sont d'abord joints. Donc *JLA* aura tendance à générer moins de tâches. Cependant, *LA* crée des arbres de précédence plus simples alors que *JLA* peut nécessiter une analyse plus complexe.

Les deux algorithmes s'exécutent en un temps linéaire par rapport au nombre de fonctions dans le DAG. L'article montre également que l'ordonnançabilité des ensembles de tâches produits par les deux algorithmes est équivalente si les surcoûts liés à l'utilisation de boîtes aux lettres et de tableaux noirs sont ignorés. Ainsi, en termes d'ordonnançabilité, ils sont comparables.

En résumé, *LA* et *JLA* adoptent différentes approches pour le placement des fonctions sur des tâches, *LA* se concentrant sur la simplicité tandis que *JLA* essaie de minimiser les tâches. Le concepteur peut choisir entre les compromis d'une précédence plus simple contre moins de tâches.

4.4.2 Comparaison avec l'état de l'art

Afin d'illustrer la comparaison entre les algorithmes LA et JLA, nous reprenons ici l'exemple de DAG présenté dans la section IV-B de l'article de référence (voir figure 4.18). Ce DAG comporte 2 événements externes e1 et e2 et 7 fonctions F1 à F7. Les temps d'exécution pire-cas et les échéances relatives des 3 chaînes fonctionnelles sont détaillés dans le tableau 4.4.

Il est à noter que dans leur exemple, les auteurs ne fournissent pas d'informations précises sur la nature périodique ou sporadique des événements en entrée, ni sur leurs périodes ou intervalles minimaux d'activation. Or, comme nous avons déjà montré précédemment, distinguer clairement les événements périodiques des événements sporadiques peut conduire à des résultats substantiellement différents en termes de délais de bout en bout. De plus, les auteurs se contentent d'indiquer que les événements sont "périodiques ou sporadiques", sans considérer l'impact de cette différence. Par conséquent, notre comparaison entre les algorithmes *LA* et *JLA* et *RYM* sur cet exemple se concentrera principalement sur leur stratégie de placement des fonctions sur les tâches générées ainsi que sur le nombre et les communications entre ces tâches, mais ne pourra pas pleinement évaluer leur performance temporelle.

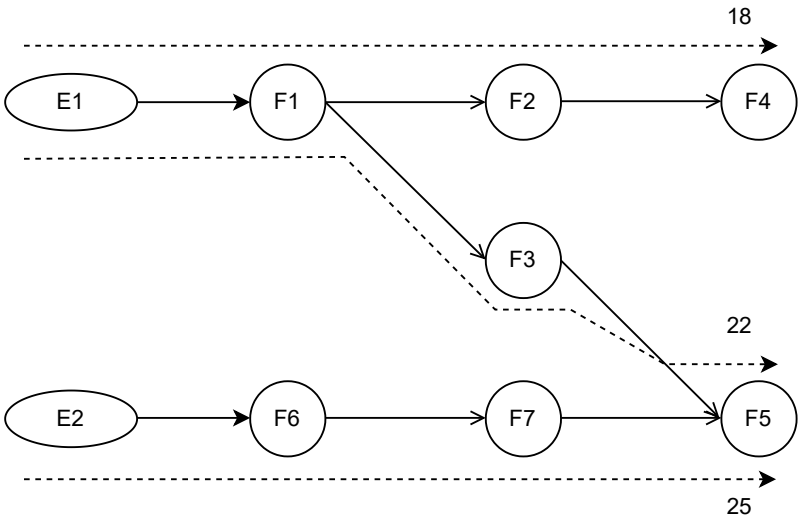

Figure 4.18: Le modèle fonctionnel utilisé dans l'article pour illustrer les algorithmes LA et JLA [BLD05].

fonction	WCET	Chaîne fonctionnelle
F1	6	
F2	3	
F3	3	
F4	1	C1=[e1,F1,F2,F4] : $\Delta 1 = 18$
F5	4	C2=[e1,F1,F3,F5] : $\Delta 2 = 22$
F6	2	C3=[e2,F6,F7,F5] : $\Delta 3 = 25$
F7	3	

Tableau 4.4: Caractéristiques de l'exemple de [BLD05]

Algorithme LA

L'application de l'algorithme LA sur le DAG génère 5 tâches (τ_1 à τ_5) comme illustré dans la figure 4.19. Chaque tâche contient les fonctions suivantes :

- τ_1 contient uniquement la fonction $F1$, qui est le successeur direct de l'événement externe e1.
- τ_2 regroupe les fonctions $F2$ et $F4$.
- τ_3 contient la fonction $F3$, autre successeur de $F1$ dans le chemin C2.
- τ_4 contient la fonction $F5$.
- τ_5 fusionne les fonctions $F6$ et $F7$ appartenant à la fonction C3 déclenché par l'événement e2.

On observe que LA génère systématiquement une nouvelle tâche pour chaque branche de successeur à partir d'une fonction.

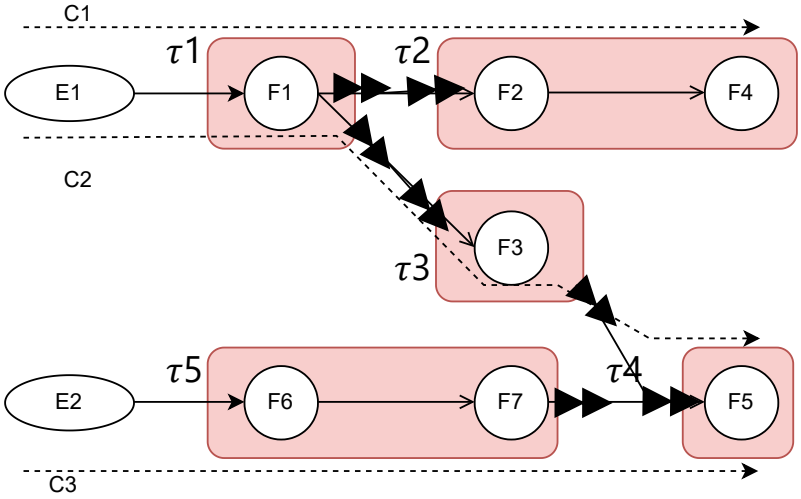

Figure 4.19: Résultat du mapping selon LA.

Algorithme JLA

L'algorithme JLA produit seulement 4 tâches (τ_1 à τ_4) à partir du même DAG, comme le montre la figure 4.20. Voici la composition des tâches :

- τ_1 fusionne les fonctions $F1$, $F2$ et $F4$, en choisissant d'abord le successeur appartenant à la chaîne C1 ayant l'échéance $\Delta 1$ la plus courte.
- τ_2 contient la chaîne $F3$, successeur de $F1$ sur la chaîne C2.
- τ_3 contient $F5$.
- τ_4 regroupe les fonctions $F6$ et $F7$ de la chaîne C3.

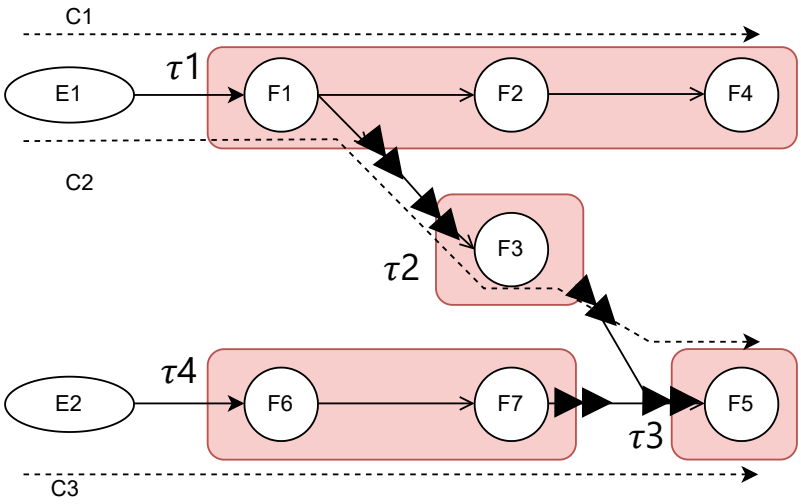

Figure 4.20: Résultat du mapping selon *JLA*.

Méthode RYM

L'application de notre méthode RYM dépend de la période maximale associée à la génération de $E1$ et $E2$. S'ils sont tous deux périodiques, RYM génère deux tâches seulement : τ_1 et τ_2, comme illustré dans la figure 4.21.

La tâche τ_1 regroupe les fonctions $F1$, $F2$, $F4$, $F3$ et $F5$. La tâche τ_2 contient les fonctions $F6$ et $F7$. Les chaînes $C1$ et $C2$ sont alors très favorisées, aux dépens de $C3$, qui paie un asynchronisme entre $F7$ et $F5$.

On observe que *RYM* a fusionné davantage de fonctions dans la tâche τ_1 par rapport aux algorithmes *LA* et *JLA* présentés précédemment. En particulier, la fonction $F5$ a été placée dans τ_1, alors que les algorithmes *LA* et *JLA* la plaçaient dans une tâche séparée τ_4 et τ_3 respectivement.

De plus, *RYM* a établi une communication asynchrone entre τ_1 et τ_2 pour transmettre les données entre les fonctions $F7$ et $F5$ appartenant à ces deux tâches.

Ainsi, sur cet exemple, *RYM* a généré un nombre de tâches sensiblement réduit en regroupant un maximum de fonctions dépendantes dans la même tâche.

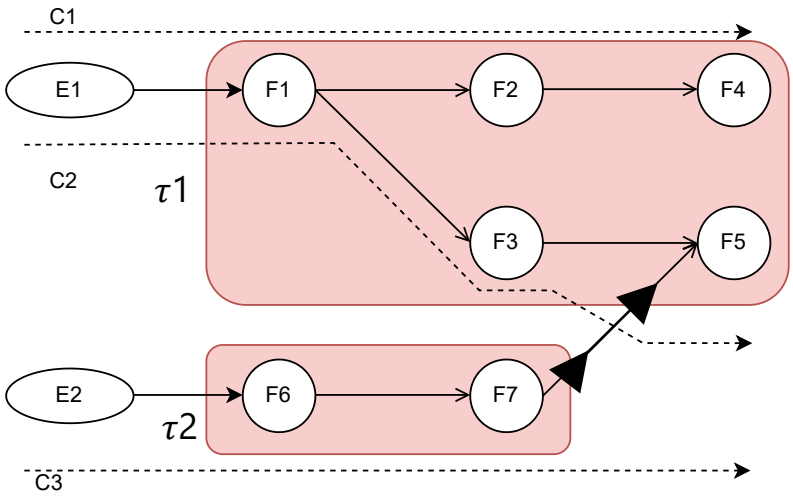

Figure 4.21: Résultat du mapping selon RYM avec une communication asynchrone.

Au cas où le rythme d'arrivée de $E1$ est sporadique ($T_{E1}^{max} = +\infty$), on ne peut pas avoir un asynchronisme entre $F7$ et $F5$, car si c'était le cas, le délai de bout en bout pire cas de $C3$ serait infini. RYM choisit donc dans ce cas de 'payer' une communication synchrone entre $F3$ et $F5$, permettant de ne payer qu'une communication synchrone (au lieu de l'asynchrone) entre $F7$ et $F5$.

Cette solution alternative génère 3 tâches comme illustré dans la figure 4.22 :

- τ_1 contient les fonctions $F1$, $F2$, $F3$ et $F4$
- τ_2 contient les fonctions $F6$ et $F7$
- τ_3 contient la fonction $F5$

La tâche τ_3 contenant $F5$ est activée via une boîte aux lettres en OU par les tâches τ_1 et τ_2.

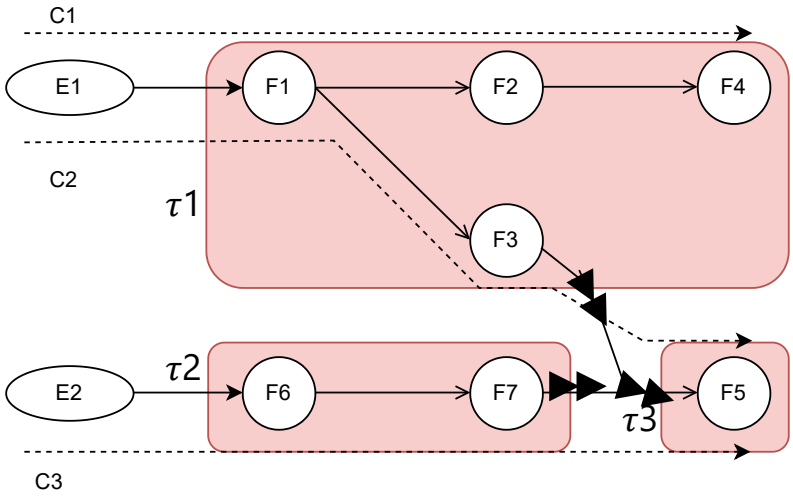

Figure 4.22: Résultat du mapping selon RYM avec une communication synchrone.

Cette étude comparative sur l'exemple de l'article [BLD05] a permis d'évaluer notre approche *RYM* face aux algorithmes *LA* et *JLA* pour la génération d'architectures logicielles temps réel.

Les résultats donnés par *RYM* sont meilleurs en termes de tâches, avec seulement 2 tâches générées dans le cas événements périodiques, et 3 dans le cas événements sporadiques, contre 5 pour *LA* et 4 pour *JLA*. Cette minimisation drastique du nombre de tâches offre des gains en mémoire utilisée et surcoût système lié à l'ordonnancement.

Nous supposons que, quelques semaines ou mois plus tard, nous connaissons enfin les WCET des fonctions, et qu'ils sont tels que donnés dans la table 4.4. La table 4.5 donne les pires délais de bout en bout calculés de façon exacte en utilisant un réseau de Petri temporel, tel que présenté dans [Lim+09], pour les méthodes *LA*, *JLA* et RYM. Pour cette analyse, nous avons supposé que les événements E1 et E2 sont sporadiques de période 50 ms, et que RYM avait donc proposé le système multitâche tel que présenté sur la figure 4.22. Nous avons aussi supposé un ordonnanceur à priorités fixes aux tâches qui donne une priorité plus élevée aux tâches faisant partie d'une chaîne fonctionnelle plus urgente. Si une tâche fait partie de deux chaînes fonctionnelles de priorités différentes, on lui assigne une priorité inférieure à son prédécesseur faisant partie de la chaîne la plus urgente, mais supérieure à son prédécesseur faisant partie d'une chaîne fonctionnelle moins urgente. Ainsi, pour RYM (voir figure 4.22), nous fixons les priorités π_i des tâches : $\pi_1 > \pi_3 > \pi_2$. Pour *LA* (voir figure 4.19) nous fixons $\pi_1 > \pi_2 > \pi_3 > \pi_4 > \pi_5$, et pour *JLA* (voir figure 4.20), $\pi_1 > \pi_2 > \pi_3 > \pi_4$.

Méthode	**Délai** $C1$	**Délai** $C2$	**Délai** $C3$
LA	10	17	26
JLA	10	17	26
RYM	10	17	26

Tableau 4.5: Pires délais de bout en bout exacts calculés à l'aide d'un réseau de Petri temporel.

Nous pouvons observer que les trois ont les mêmes délais de bout en bout pire cas exacts en priorité fixe. Cependant, le plus faible nombre de tâches donne un avantage supplémentaire en termes de surcoût mémoire. En effet, la taille de pile d'une tâche exécutant deux fonctions séquentiellement est inférieure à la somme des tailles de piles de deux tâches exécutant ces fonctions. Il en va de même pour le surcoût processeur et mémoire lié à la communication de variable entre deux tâches : Nous nous sommes limités à un seul exemple, cependant celui-ci est basé sur l'étude de cas proposée par les auteurs de *LA* et *JLA*.

4.5 Outil

La figure 4.23 présente le métamodèle de la partie fonctionnelle exprimé en Ecore, un langage de méta-modélisation similaire à UML [Fou23]. Le métamodèle est constitué des éléments suivants :

- Une classe racine nommée `FunctionalDiagram` qui représente le diagramme fonctionnel. Elle est composée de :

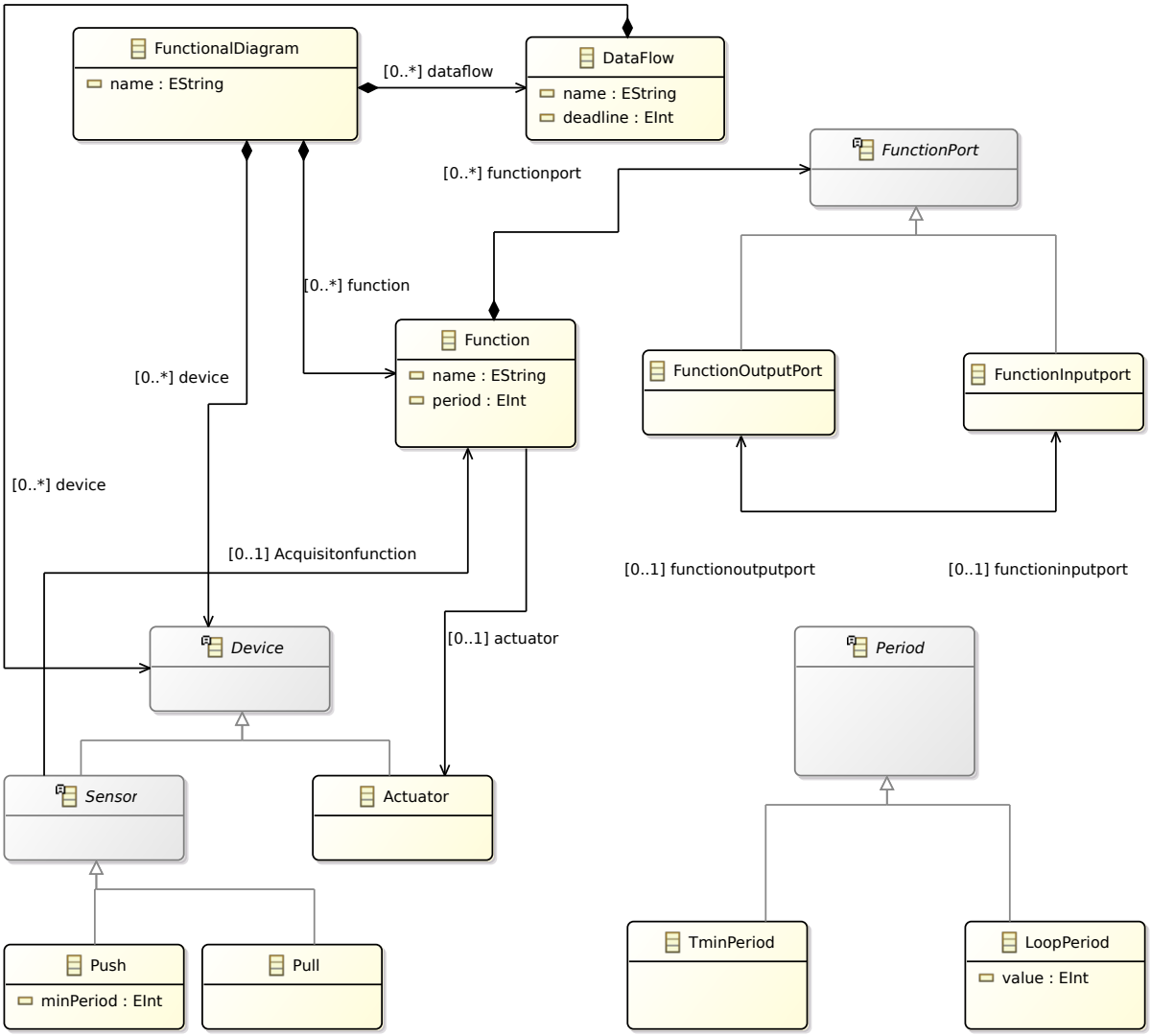

Figure 4.23: Métamodèle de la partie fonctionnelle du point de vue AADL-Like

- Un ensemble de flots de données (`Dataflow`) caractérisés par un nom et une échéance.
- La classe `Device` qui représente les éléments fonctionnels et matériels. Un dispositif peut être :
  - Un actionneur
  - Un capteur, qui peut être de type `push` ou `pull`
- La classe `Function` qui représente les fonctions. Les fonctions possèdent des ports (`FunctionInputPort` et `FunctionInputPort`) pour pouvoir échanger les données avec les autres éléments via les flots de données.

Le métamodèle présenté dans la figure 4.24 représente la partie déploiement, il décrit le modèle conceptuel associé au déploiement. Ce modèle est également exprimé en Ecore. Il étend le métamodèle fonctionnel avec les éléments nécessaires tels que l'élément `Thread` et ses caractéristiques, processeur, processus, ainsi que les éléments responsables des communications entre les threads tels que les éléments `ThreadPort` qui peuvent être de type *data*, *event* ou *event-data*.

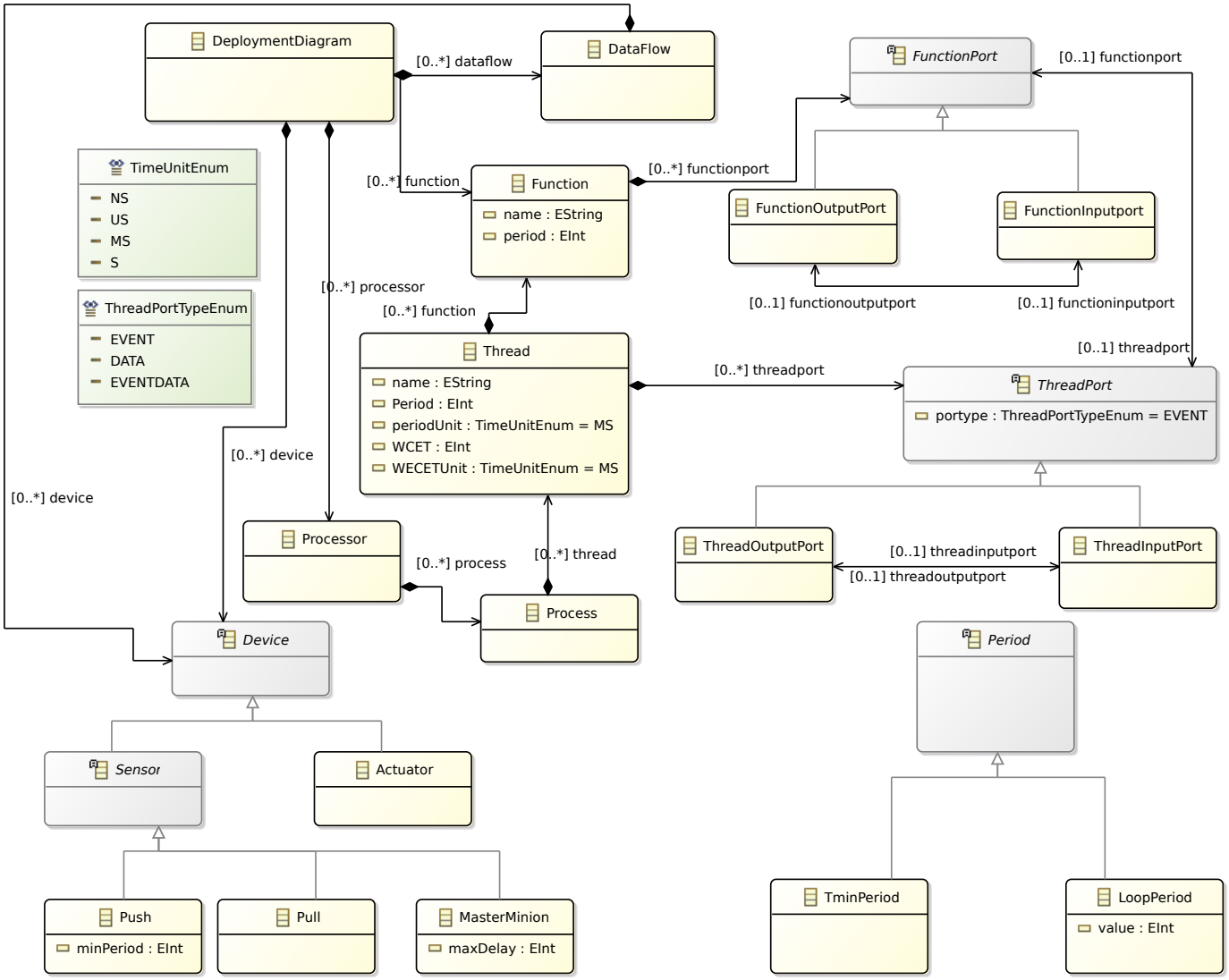

Figure 4.24: Métamodèle de la partie fonctionnelle du point de vue AADL-Like

La figure 4.25 montre l'éditeur de diagramme qui permet à l'utilisateur d'exprimer le modèle fonctionnel. À ce niveau, nous avons décidé d'enrichir le diagramme d'architecture logique de Capella en y ajoutant des propriétés temps réel, étant donné que certaines des entités de métamodèle proposées sont déjà présentes dans ce diagramme de Capella. Lorsque le concepteur sélectionne un composant, il a la possibilité d'entrer les caractéristiques spécifiques de ce dernier, telles que la "date limite de bout en bout", qui représente une propriété temps réel associée à une chaîne fonctionnelle (mise en évidence dans la figure 4.25).

Une fois que le concepteur du système a achevé la partie fonctionnelle, il peut générer directement le modèle de déploiement en suivant la méthode basée sur les rythmes. Le résultat du déploiement est ensuite visualisé dans un nouveau diagramme distinct de celui fourni en entrée. La figure 4.27 présente le résultat du déploiement au sein de l'outil. À ce stade, nous enrichissons le diagramme de l'architecture physique en introduisant de nouveaux éléments spécifiques au domaine, tels que les tâches, les communications, les processus, etc. La syntaxe graphique concrète utilisée dans ce diagramme s'inspire de la syntaxe graphique de AADL, car il s'agit d'un langage standardisé.

Le concepteur a la possibilité de modifier la conception proposée selon ses besoins. À cet effet, nous proposons une palette d'outils, représentée dans la figure 4.26, qui offre l'ensemble des composants graphiques nécessaires, en plus de l'onglet de *déploiement* dans la section des propriétés (voir la partie surlignée dans la figure 4.27).

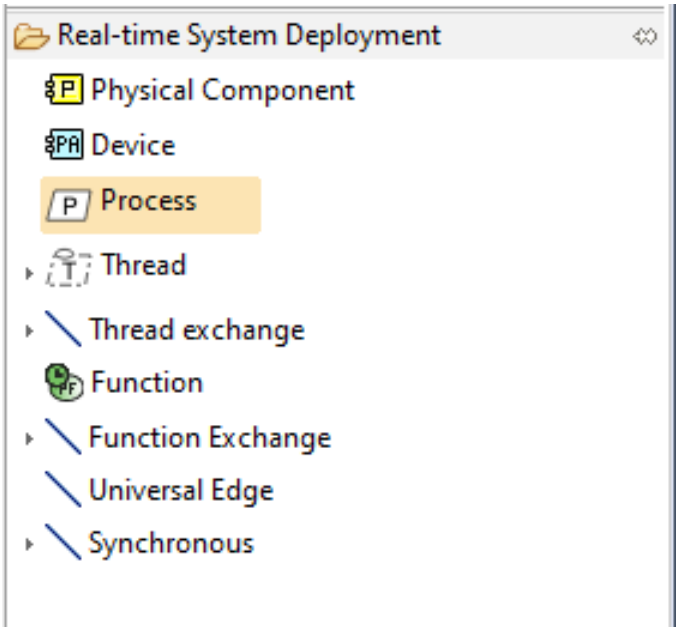

Figure 4.26: Palette d'outils.

4.6 Conclusion

Dans ce chapitre, nous avons présenté la méthode des rythmes, une méthode pour le pré-déploiement des systèmes temps réel critiques lors des phases initiales de conception du système. Cette technique permet de réduire le nombre de tâches et de contrôler le

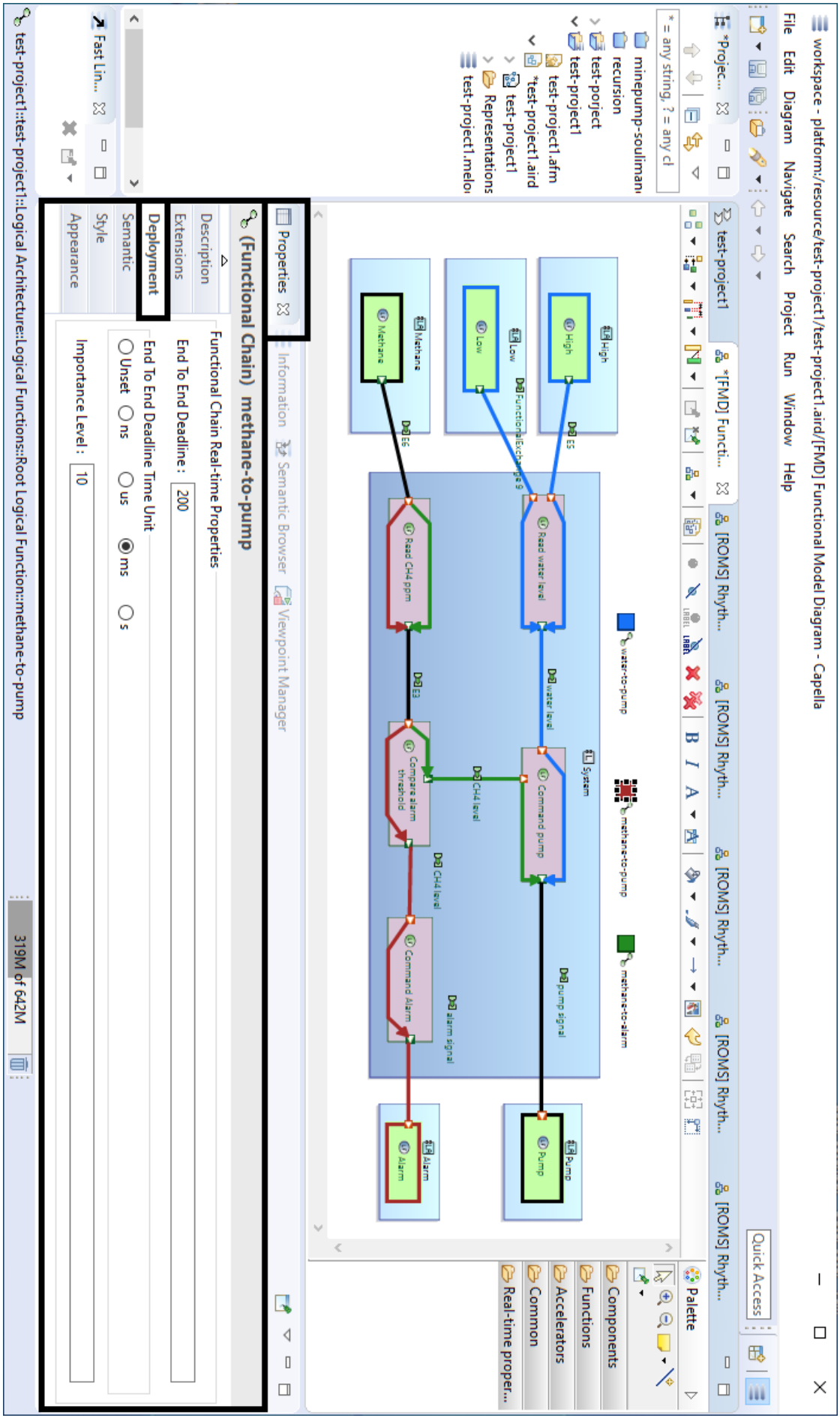

Figure 4.25: Diagramme de l'architecture logique de Capella enrichi pas les propriétés non-fonctionnelles

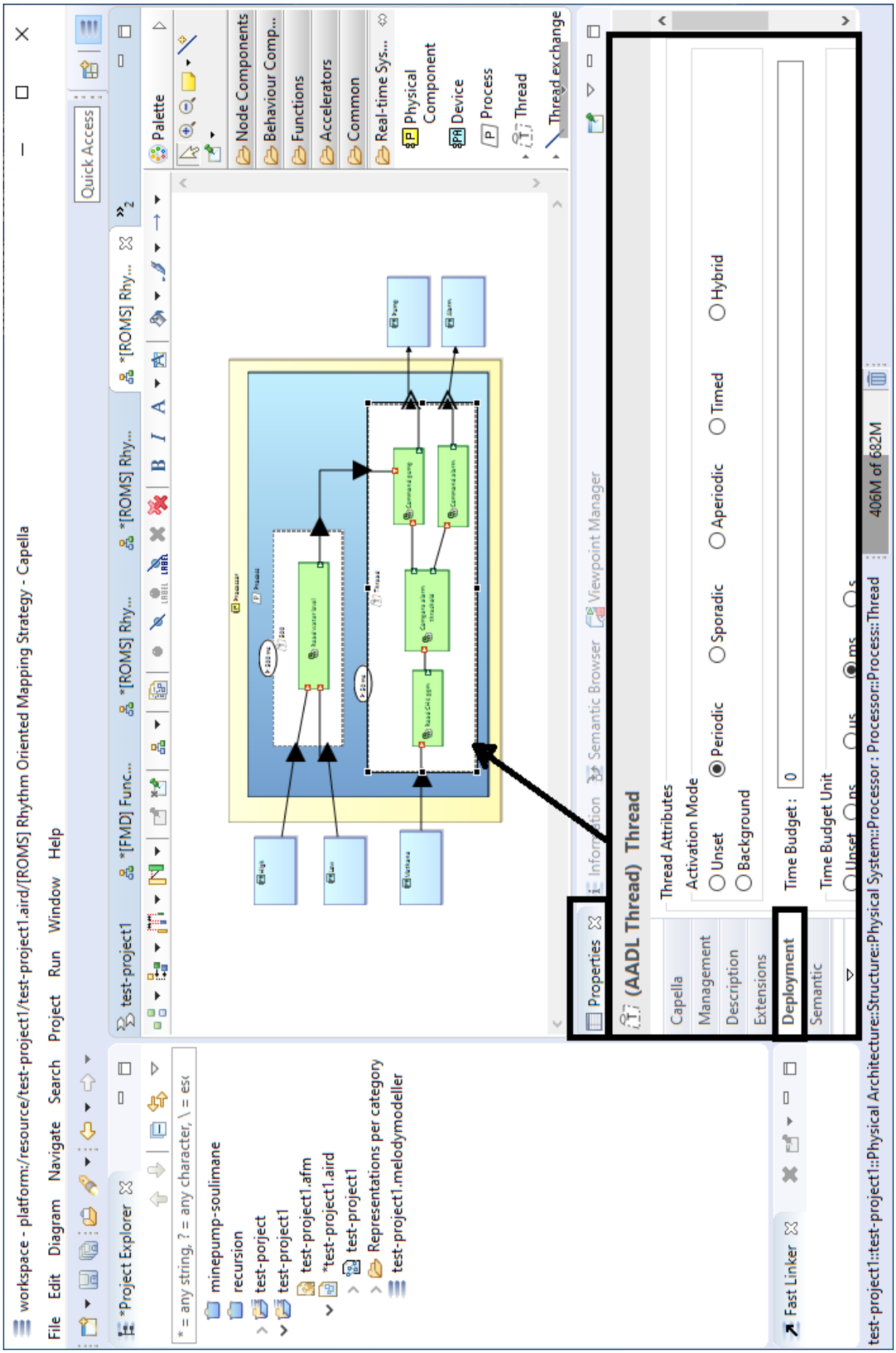

Figure 4.27: Le résultat du déploiement sous Capella

pire délai de bout en bout sur les chaines fonctionnelles, en le minimisant sur les chaînes fonctionnelles urgentes. La méthode est pessimiste, et la conception de pré-déploiement est correcte par construction et peut être affinée. De plus, la méthode des rythmes est démontrée comme étant indépendante de tout système d'exploitation, comme le montre une étude d'analyse d'ordonnançabilité basée sur le temps de réponse en utilisant deux politiques d'ordonnancement différentes : priorité fixe et EDF.

Il est important de noter que la méthode des rythmes utilise les informations généralement existantes au moment du mapping des fonctions sur les tâches, contrairement à la plupart des méthodes de l'état de l'art. Sa simplicité permet aussi au concepteur de l'appliquer à la main, et il peut au besoin en faire des variantes. Ainsi, si minimiser le délai de bout en bout de la chaîne fonctionnelle la plus urgente nuit fortement à une chaîne moins urgente, il est facile de remplacer le prix de l'asynchrone payé par la seconde chaîne, en partageant le coût et en payant le prix synchrone sur les deux chaînes. La méthode, par la prise en compte des périodes maximales, permet aussi d'intégrer des éléments de tolérance aux fautes.

Nous avons présenté également un outil basé sur les modèles pour spécifier les propriétés non fonctionnelles en plus des artefacts fonctionnels et générer un modèle de déploiement selon la méthode des rythmes. Notre méthode vise à produire un déploiement pour une large gamme de systèmes d'entrée, indépendamment des propriétés non fonctionnelles spécifiques ou des techniques d'analyse. Enfin, nous avons déroulé notre méthode sur un exemple réaliste et présenté le résultat de déploiement généré, ainsi que l'outil fourni.

CHAPITRE 5

Personnalisation du logiciel d'autopilotes de drone à travers un framework de rétro-ingénierie

5.1 Introduction

Les véhicules aériens sans pilote (UAV), également connus sous le nom de drones, sont de plus en plus répandus et utilisés dans divers domaines. De nombreuses petites et moyennes entreprises développent des drones personnalisés, qui offrent des fonctionnalités matérielles et/ou logicielles innovantes. Généralement, les solutions proposées utilisent des autopilotes open source prêts à l'emploi (Off-The-Shelf). En réalité, un autopilote est un générateur d'autopilote visant différentes plateformes matérielles. Un autopilote est un système embarqué temps-réel complexe. Afin d'offrir des fonctionnalités innovantes, les dronistes doivent personnaliser et étendre les fonctionnalités de l'autopilote retenu. Cependant, la compréhension de l'architecture et du fonctionnement interne des autopilotes open-source est difficile en se basant uniquement sur leur code source. En effet, le tableau 5.1 donne une indication de l'étendue des efforts de développements investis dans les autopilotes open source. Il est à noter qu'un autopilote visant une plateforme donnée utilise généralement moins de 10% des fichiers sources trouvés sur le dépôt de sources d'un autopilote. Bien que les autopilotes proposent des moyens documentés d'ajouter des modules personnalisés, par exemple grâce à des intergiciels tels que ROS, la granularité temporelle des modules est élevée par rapport à la vitesse de fonctionnement du cœur de l'autopilote. Paparazzi propose d'intégrer des modules, qui pourront être appelés directement par la boucle de régulation interne d'attitude, cependant, comme nous le verrons, cela présente un risque au niveau comportement temporel. Lorsque le module à ajouter doit s'exécuter à une fréquence de l'ordre de celle du cœur d'autopilote, il est nécessaire de fournir d'une part les moyens de comprendre le fonctionnement de l'autopilote, et aussi de faciliter l'analyse de l'autopilote modifié, en particulier son analyse de performances.

	Paparazzi	**PX4**
C files	2015	526
C LOC	513517	182009
C++ files	137	1025
C++ LOC	402617	287175
Makefiles	366	534
Makefile LOC	56815	26663
Config files	1260 (XML)	260 (Kconfig)
Config LOC	160656	1819

Tableau 5.1: Nombre de fichiers et nombre total de lignes de code dans deux projets phares d'autopilotes open source en novembre 2022.

Dans ce chapitre, nous présentons un framework pour extraire, modéliser et analyser l'architecture logicielle d'un autopilote afin de faciliter sa personnalisation. Nous commençons par présenter les concepts spécifiques développés pour représenter les autopilotes. Nous décrivons ensuite le cas d'étude de l'autopilote open-source Paparazzi et ses spécificités. Puis, nous détaillons le framework proposé pour extraire un modèle de l'architecture à partir du code source. Ce modèle permet de représenter visuellement la décomposition fonctionnelle et multitâche, ainsi que les chaînes fonctionnelles. Nous présentons comment ce framework a été appliqué sur Paparazzi. Enfin, nous illustrons son utilisation pour adapter une technique d'ordonnancement afin d'intégrer des modules personnalisés.

Le framework présenté dans ce chapitre vise à faciliter la compréhension, l'analyse et l'adaptation des autopilotes open-source pour les drones. Le framework d'extraction de modèle permet de faire le lien entre le code source et les outils de modélisation et d'analyse pour la conception de systèmes embarqués.

5.2 État de l'art sur la visualisation logicielle et la rétro-ingénierie

5.2.1 Analyser le code pour faire de la rétro-ingénierie

Avant de visualiser le code, il est nécessaire d'en extraire les propriétés d'intérêt. Cette tâche de rétro-ingénierie est accomplie en utilisant des outils d'analyse de code.

Les outils d'analyse de code sont généralement classés en analyse statique (analyser le code tel quel), analyse dynamique (analyser les traces d'exécution sur des programmes instrumentés), ou analyse historique (analyser les changements successifs effectués par les développeurs). Dans notre cas, nous traitons du code C, pouvant utiliser des primitives multitâche spécifiques au système d'exploitation utilisé, et qui combine des primitives système (ex : pour la gestion des tâches POSIX, pilotes de périphériques) et du code applicatif. À partir de ce code, nous visons à obtenir des caractéristiques temps-réel (ex : périodicité) et des dépendances fonctionnelles entre tâches (pour l'analyse de flot de données).

Bien qu'il soit possible que d'autres outils puissent être utilisés pour réaliser cette tâche sans remettre en question le framework, nous n'avons pas trouvé de tels outils. Dans la table 5.2, nous listons différents types d'outils d'analyse de code proches de nos besoins, mais n'ayant pas été pleinement satisfaisants. À l'exception de SolidSX [RTV10], que nous avons retenu, car il fournit une vue de graphe d'appels cohérente avec notre travail.

5.2.2 Visualisation

De nombreux outils de rétro-ingénierie et de visualisation logicielle (parfois combinés ou séparés) sont disponibles dans les domaines académique et commercial (voir les études [Kos03; CDC11]).

Ces outils peuvent être utilisés pour calculer et illustrer divers indicateurs tels que le nombre de lignes de code, le nombre de méthodes, la complexité, l'organisation du code, etc.

Par exemple, [LD03] propose une approche pour représenter les entités logicielles du code source orienté-objet et leurs relations. L'accent est mis sur la fourniture de vues graphiques de différents indicateurs logiciels. Dans l'article [WL07], le code source orienté objet est représenté graphiquement comme une ville navigable en 3D. Les classes sont représentées comme des bâtiments dans la ville, et les packages sont représentés comme des districts dans lesquels se trouvent ces bâtiments.

Bien que ces approches se concentrent sur la visualisation de code orienté objet, notre objectif dans ce travail n'est pas d'offrir une vue orientée objet, mais plutôt de fournir une vue fonctionnelle d'un code qui n'est pas (ou très peu) orienté objet.

Outil	Langage(s)	Licence	Usage
CPAChecker [BK11]	C	Académique	Analyse statique : vérification formelle
Synopsis [Inc22]	C/C++/Java et C#	Commercial	Sécurité
cppcheck [Tea]	C/C++	Open source	Analyse statique : détection de bugs et de comportements indéfinis
Frama-C [Cuo+12]	C (extension C++)	Académique	Analyse statique : aucune extension pour l'analyse de flots de données
PVS-Studio [LLC]	C/C++/Java et C#	Commercial	Analyse statique
shinyprofiler [Gra]	C/C++/Lua	Open source	Profileur
SolidSX [RTV10]	C/C++	Commercial	Offre une vue graphique des appels
VectorCast [Gmba]	C/C++	Commercial	Test automatisé de logiciels embarqués
Zoom [Gmbb]	C/C++/ObjC/ Fortran/Assembly	Commercial	Profileur statistique Linux
OovAide [JZ16]	C++/Java	Open source	Analyse automatisée des dépendances du code source C++

Tableau 5.2: Outils d'analyse de code

Cette partie nécessite un DSL pour représenter le logiciel embarqué, en particulier les concepts de l'architecture logicielle des autopilotes, englobent à la fois les contraintes fonctionnelles et non-fonctionnelles, ainsi que l'architecture. Ce DSL devrait permettre une analyse d'ordonnançabilité. À notre connaissance, aucun outil existant n'a été proposé pour répondre à ces exigences.

5.3 Conception des autopilotes des UAV

Les méthodes de conception utilisées pour créer un autopilote personnalisé diffèrent de celles utilisées pour les autres systèmes embarqués. Contrairement à la plupart des autres systèmes, qui suivent une approche top-down dans leur cycle de vie de conception, un autopilote personnalisé nécessite une approche différente. Avec une approche top-down, la décomposition fonctionnelle peut être dérivée indépendamment du matériel en fonction des exigences. Les fonctions sont ensuite mappées sur des entités exécutables au niveau bas, qui correspondent à des processus et tâches, puis mappées sur un processeur, avec ou sans système d'exploitation.

Cette approche a été utilisée dans plusieurs domaines de l'informatique embarquée depuis des décennies. De nombreuses méthodologies sont basées sur cette approche top-down, y compris l'analyse structurée pour le temps réel (SA/RT) [Ros77] dans les années 1980, l'architecture pilotée par modèle (MDA) [Bro04] lancée par l'Object Management Group (OMG) au début des années 2000, et la méthode ARCADIA [Tha23] outillée par Capella dans les années 2010. L'approche top-down est également utilisée dans la

norme automobile AUTOSAR [aut23], ainsi qu'en aéronautique avec les normes DO-178C [Bro11].

La personnalisation requiert d'abord le choix d'une instance d'autopilote, qui nécessite une plateforme matérielle compatible avec l'autopilote choisi, un système d'exploitation compatible, et des capteurs et actionneurs qui sont également compatibles avec la plateforme et l'autopilote. Le cadre (avion, quadricoptère en croix ou en X, hexacoptère, etc.) peut être fait sur mesure ou basé sur un cadre existant, et de la valeur peut être ajoutée à l'UAV en personnalisant le cadre ou en ajoutant de nouvelles fonctions qui ne sont pas implémentées dans l'autopilote open source, ou en incorporant du matériel spécifique.

Ensuite, contrairement à une approche descendante classique, le processus de développement consiste à étendre l'instance d'autopilote pour l'adapter aux pièces personnalisées. Bien que certains autopilotes puissent facilement intégrer de telles extensions, le défi réside dans l'intégration de certains modules personnalisés qui peuvent non seulement être difficiles à intégrer, mais également nuire au bon fonctionnement de l'autopilote d'origine, ce qui peut potentiellement le faire dysfonctionner.

Dans le cas d'un module relativement autonome vis-à-vis de l'autopilote, qui va interagir avec lui avec des délais de l'ordre de plusieurs dizaines voire centaines de millisecondes, l'autopilote peut être vu comme un pilote de périphérique de haut niveau, une brique *Component Off-The-Shelf* (COTS). Cependant, dans le cas où les modules additionnels sont intégrés au cœur de l'autopilote, avec des délais attendus de l'ordre de quelques millisecondes, voire centaines de microsecondes, entre le calcul d'état interne et l'exécution d'une fonction personnalisée, alors l'autopilote peut être perçu comme un *Modifiable Off-The-Shelf* (MOTS). Afin de mieux comprendre ce qui est en jeu, et ce que l'on trouve dans le cœur d'un autopilote, nous présentons ci-après les fonctions type d'un autopilote, avec les fréquences typiques observées sur les autopilotes. Nous présentons ensuite la façon dont Paparazzi est implémenté pour répondre à ces exigences de fréquences.

5.4 Architecture logicielle abstraite d'un autopilote

La figure 5.1 est essentielle pour comprendre le fonctionnement d'un autopilote, en particulier pour un drone. Le niveau minimal d'autonomie d'un drone est illustré dans les cases bleu clair. Au cœur de ces fonctions, indiqué dans la boîte en pointillé, il y a trois fonctionnalités : les deux premières, l'estimation d'état et le contrôle de l'attitude, sont nécessaires pour stabiliser le drone. La troisième correspond au bloc Adaptateur géométrique, qui adapte les sorties du régulateur, est responsable d'adapter la commande calculée aux différents types de cadres. Noter que la fréquence attendue de cette boucle principale peut être très élevée, jusqu'à 3 kHz sur certaines voilures tournantes.

Un autopilote a des modes de vol intégrés pour prendre en charge différents niveaux et types de stabilisation de vol ou de missions dédiées. Dans certains modes de vol, tels que les modes de vol assistés, les deux blocs supérieurs centraux en violet (contrôleurs de position et de trajectoire) ne sont pas utilisés et l'opérateur envoie directement des points de consigne de haut niveau réguliers (par exemple, vitesse horizontale/verticale et angle de roulis) via la liaison montante de télémétrie. Les modes qui définissent un point cible nécessitent un contrôleur de position. Son rôle est de calculer des points de consigne de haut niveau pour le contrôleur d'attitude, en fonction de l'attitude et de la position

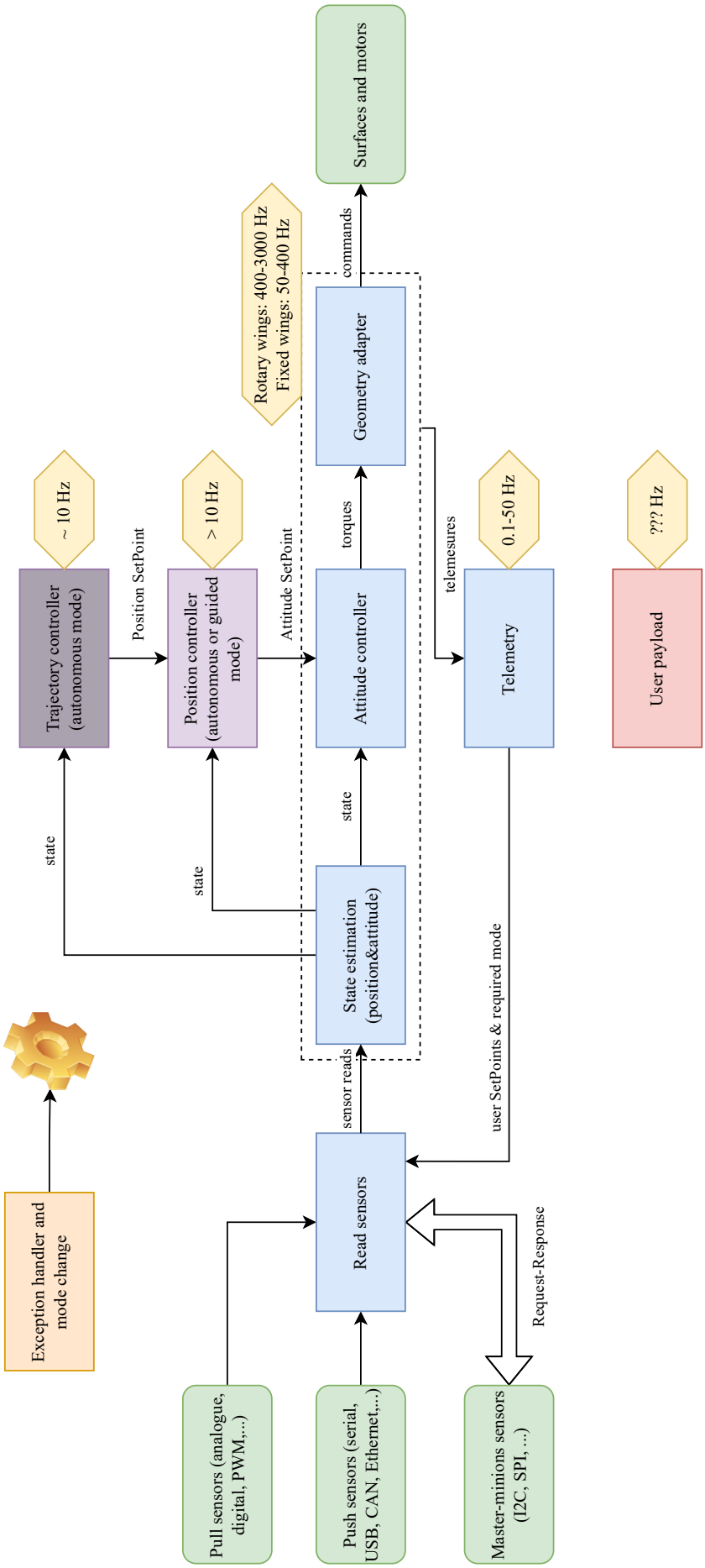

Figure 5.1: Principaux composants d'un autopilote

actuelles par rapport au point de passage ciblé. En présence d'obstacles ou lorsqu'un chemin complexe doit être suivi, un autre bloc de niveau supérieur, chargé de calculer le chemin, est exécuté. Ce bloc fournit des points de consigne au contrôleur de position, qui à son tour fournit des points de consigne au contrôleur d'attitude. On peut voir sur la figure 5.1 que les fréquences sont généralement faibles pour ces deux blocs par rapport à la fréquence de la boucle principale, véritable cœur de l'autopilote.

Dans certains cas, des modules personnalisés supplémentaires sont exécutés en interaction avec l'autopilote. Dans certains cas, ces modules personnalisés supplémentaires peuvent nécessiter de courts délais entre l'estimation d'état et leur exécution, et peuvent même nécessiter d'être placés entre le contrôle d'attitude et l'actionnement. Ces modules personnalisés sont représentés par le bloc " Charge utile de l'utilisateur " rose de la figure 5.1. Leur fréquence et leurs interactions avec les autres blocs varient en fonction de leur nature.

5.5 Exemple motivationnel

La valeur ajoutée d'un nouveau drone réside dans les fonctions personnalisées qui ne sont pas initialement prévues par les concepteurs de l'autopilote MOTS. Certaines de ces fonctions n'exigent pas de délais courts entre l'estimation de l'état et leur exécution. Par exemple, il peut s'agir d'une fonction personnalisée qui n'affecte pas l'autopilote (par exemple, une station météo) ou qui modifie uniquement le contrôleur de trajectoire, qui s'exécute à une fréquence de l'ordre de 100 millisecondes ou plus. Une telle fonction peut être hébergée sur une carte compagnon ou dans une tâche ou même un processus sur la même carte que celle exécutant l'autopilote s'il y a un système d'exploitation. Dans ce cas, la communication entre le cœur de l'autopilote et cette fonction peut utiliser des moyens de communication middleware tels que ROS [Qui09] ou MAVLink [Kou+19] via UDP [Pos80]. Si la fonction personnalisée doit être exécutée avec une fréquence plus courte, et que les délais sont importants, par exemple, si elle doit être exécutée à la même fréquence que le contrôleur d'attitude ou le contrôleur de position, alors les middlewares peuvent introduire trop de délais pour être pris en compte. Ainsi, ce module personnalisé doit être intégré dans le cœur de l'exécution cyclique. Des exemples de fonctions pouvant avoir un impact sur le contrôleur de position lui-même sont le contrôle d'essaim et la barrière potentielle. Les fonctions impactant le contrôleur d'attitude pourraient être, par exemple, le maintien d'un objet mobile libre au-dessus d'un drone à ailes rotatives.

Pour déterminer quels leviers peuvent être utilisés pour que les modules personnalisés ne rallongent pas la durée de la boucle centrale au-delà de la période T de cette boucle, nous considérons un module personnalisé fictif qui exécute plusieurs fonctions périodiques nécessitant un calcul intensif de l'ordre de $\frac{T}{4}$. À noter que si plus d'une fonction périodique est appelée dans la même boucle, la durée d'exécution de la partie non visible de la boucle principale sera nécessairement supérieure à $\frac{T}{2}$ et retardera l'exécution de la boucle principale suivante. Les cas suivants sont envisagés, en supposant une fréquence de boucle principale de 500 Hz, donc $T = 2$ ms. Toutes les fonctions suivant l'estimation de l'état reçoivent $\frac{T}{2} = 1$ ms pour être exécutées, avant le début de la prochaine itération de la boucle centrale.

Cas 1: 11 fonctions périodiques ayant une même période de 10 s

Cas 2: 3 fonctions périodiques ayant des périodes de 6, 60 et 66 ms

Cas 3: 2 fonctions périodiques ayant des périodes de 1 et 10 s

Pour éviter les instants critiques, Paparazzi utilise une approche naïve pour ajuster les décalages des fonctions périodiques : elles sont triées par périodes croissantes et numérotées de 0 à $n-1$. Chaque fois qu'une nouvelle fonction périodique de numéro i et de période T_i (exprimée en périodes T de la boucle centrale) est ajoutée à la boucle principale, son décalage est de $\frac{i}{10} \times T_i$, ajoutant 10 % de la période à chaque fois. De toute évidence, cette stratégie peut être facilement contournée, par exemple, dans le cas 1, les première et onzième fonctions coïncident. Cela peut être observé sur le graphique supérieur de la figure 5.2, où la durée de la partie commande de la boucle centrale est affichée. Chaque fois qu'elle dépasse une milliseconde, la boucle centrale actuelle déborde sur la période suivante. Cela peut avoir un impact sur la stabilité, car le contrôleur d'attitude sous-jacent suppose une exécution à la fréquence choisie. Même avec moins de fonctions périodiques, comme les cas 2 et 3, l'approche d'ajustement de décalage naïve peut être contournée. Pour faire face à cela, nous devrions être en mesure d'utiliser un outil permettant d'éviter les instants critiques entre les fonctions périodiques. Cependant, cet outil, basé sur GCD+, une heuristique puissante pour choisir des décalages dans les systèmes à décalage libre, ne peut pas directement traiter le code C/C++ car il nécessite un modèle d'entrée du système.

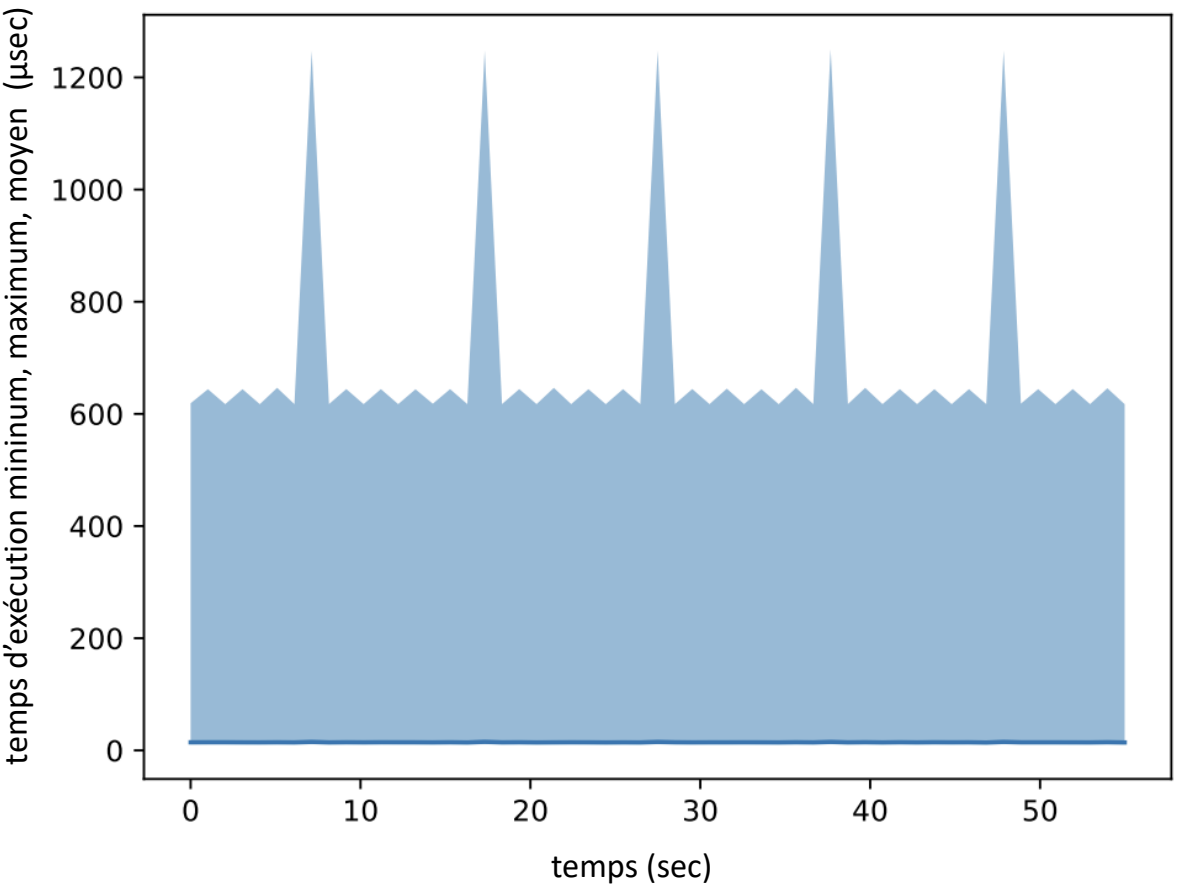

Figure 5.2: Temps d'exécution (temps minimum et maximum) de la phase de commande de la boucle principale pour le cas 1 en utilisant les décalages natifs de Paparazzi. Chaque point représente une seconde d'exécution. La zone bleu clair indique l'intervalle min/max. La ligne bleu foncé représente la moyenne (très proche du minimum dans ce cas, montrant que généralement l'autopilote utilise peu de ressources processeur).

5.6 Architecture de Paparazzi

5.6.1 Autopilotes pour UAV

Les autopilotes open source sont des frameworks hautement adaptables qui permettent de générer des autopilotes spécifiques. Ces autopilotes peuvent être étendus avec des logiciels et des matériels personnalisés, les logiciels personnalisés utilisent l'autopilote comme un COTS ou comme un MOTS selon le degré d'interaction nécessaire avec le cœur de l'autopilote. Lorsque l'interaction avec le cœur doit se faire à fréquence élevée, pour modifier un autopilote comme un MOTS, il faut une connaissance approfondie de l'informatique et du code source de l'autopilote. L'intégration de nouveaux capteurs est relativement simple, car il existe généralement un processus bien documenté pour intégrer

des modules qui se conforment à une interface spécifique. Cependant, la modification du comportement de l'UAV, en particulier toute fonctionnalité entre l'estimation de l'état et l'actionnement, peut-être une tâche laborieuse avec des risques potentiels pour la sécurité.

En figure 5.3, nous pouvons voir la boucle centrale d'un système de pilote automatique typique. La boucle fonctionne à une fréquence déterminée par la stabilité inhérente de l'aéronef. Un point de consigne est établi pour définir l'état souhaité de l'UAV. Ce point de consigne peut être déterminé par l'utilisateur dans les modes de vol assistés, ou par la boucle de guidage de vol dans les modes plus autonomes, qui garantit que la trajectoire de l'UAV respecte un plan prédéterminé. Les capteurs fournissent des lectures qui permettent d'estimer l'état réel de l'UAV, qui peut inclure les vitesses angulaires sur chacun des axes tridimensionnels, les vitesses horizontales aériennes et/ou terrestres, la vitesse de montée, ainsi que les valeurs angulaires réelles sur les axes tridimensionnels dans certains cas. Ces informations sont fusionnées, généralement par filtre de Kalman étendu (EKF), afin de calculer l'état de l'aéronef.

L'erreur entre l'état estimé et le point de consigne est calculée et peut être corrigée par plusieurs types de correcteurs, tels que les PID (Proportional, Integrator, Derivative) ou les correcteurs PD. Plus récemment, les contrôleurs INDI (Incremental Non-Linear Dynamic Inversion) ont également été utilisés, car ils présentent une meilleure stabilité. Ces contrôleurs calculent les couples à appliquer à chaque axe, qui sont ensuite convertis en positions individuelles des surfaces ou en commandes des rotors par un processus de mixage. Cela permet au contrôle de rester relativement indépendant de la géométrie du drone.

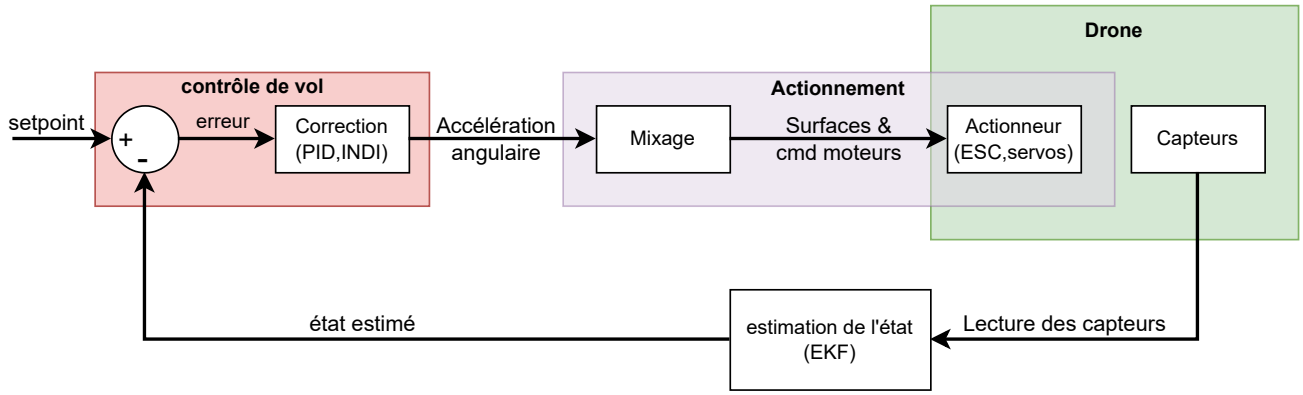

Figure 5.3: Boucle de stabilité interne dans un autopilote

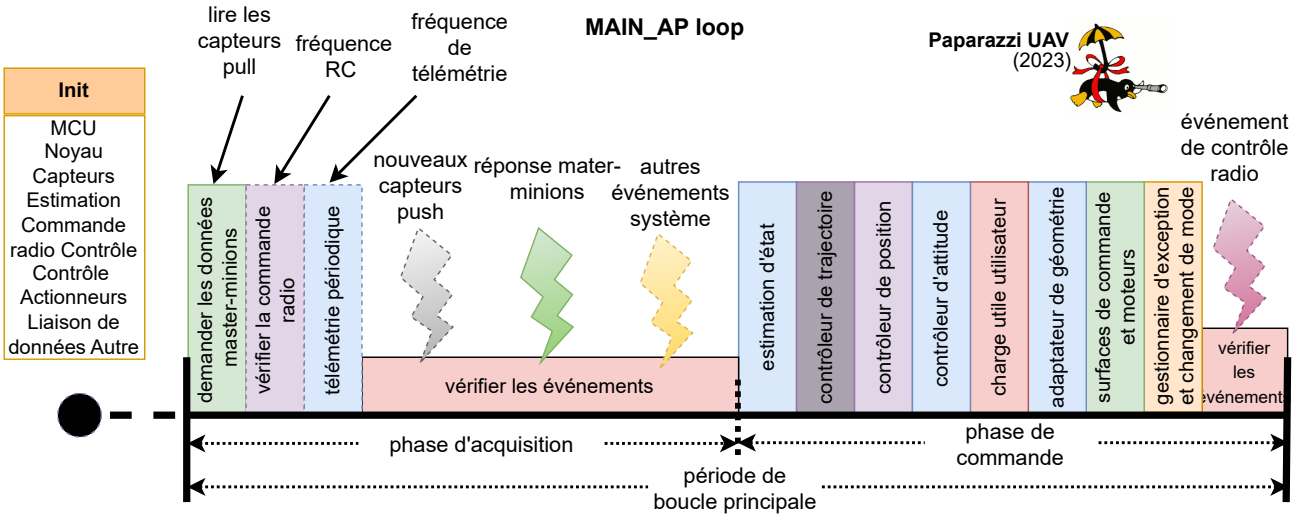

Figure 5.4: Boucle principale de l'autopilote Paparazzi

Certains autopilotes, comme PX4, ne peuvent être exécutés que sur un système d'exploitation compatible POSIX, tandis que d'autres, comme Paparazzi et Ardupilot, peuvent cibler soit un RTOS plus léger, soit une plateforme sans système d'exploitation dite alors bare-metal. Cela peut être très utile si l'autopilote est utilisé comme un autopilote de sécurité sur un système multiprocesseur hétérogène sur une puce (HMPSoC). On peut imaginer un autopilote complexe comprenant des fonctions de haut niveau, s'appuyant sur l'apprentissage profond pour le traitement d'images, et exécuté sur un RTOS basé sur Linux (par exemple, pour inclure des bibliothèques telles que TensorFlow), sur un cluster de cœurs rapides. Ce type d'autopilote est difficile, voire impossible, à certifier par rapport à tout standard de sûreté. Pour faire face à toute défaillance, un autopilote bare-metal léger peut être exécuté sur un microcontrôleur plus lent du type HMPSoC. Cet autopilote peut prendre le contrôle en cas de défaillance de l'autopilote principal, ce qui peut faciliter le processus de certification.

Sur une plateforme bare-metal, le modèle de programmation habituel repose sur une boucle d'arrière-plan pour l'exécution du système, tandis que les routines de service d'interruption stockent les trames de capteurs sporadiques dans des buffers circulaires. D'un point de vue ordonnancement, cela peut être vu comme un exécutif cyclique non préemptif. Il est exécuté soit aussi vite que possible, soit un délai est inséré entre deux exécutions successives pour une période spécifique T à utiliser. Cette période est dérivée de la fréquence requise du bloc de cœur d'autopilote exprimée dans la figure 5.1. Vérifier si des données de capteurs entrantes provenant de capteurs sporadiques (push) sont disponibles, revient à regarder si des données sont disponibles dans le buffer. Si tel est le cas, l'autopilote exécute la fonction correspondante de traitement des données. Une telle fonction, d'un point de vue ordonnancement, peut être considérée comme une fonction sporadique non préemptif ayant une période multiple de T. Les capteurs scrutés, de type pull, sont lus à leur période requise, qui doit être un multiple de T. Les capteurs maître-sbires (maître-minions ou parfois aussi appelé *master-slave* en anglais) sont également interrogés à la période souhaitée (également un multiple de T), mais leur réponse est retardée et ne peut pas être attendue de manière synchrone sans retarder l'exécutif cyclique sous-jacent. En conséquence, certains traitements qui peuvent être effectués avant l'estimation d'état peuvent être exécutés entre la requête et la réponse. Ainsi, par exemple, la lecture de la télémétrie d'entrée peut être traitée entre requête et réponse des

capteurs I2C.

Pour mettre en œuvre cette stratégie, Paparazzi divise la période d'exécution T de la boucle principale en deux parties, comme le montre la figure 5.4. La contrainte est que chaque transaction maître-sbires doit être réalisée pendant la première partie, et que toutes les données des capteurs sont collectées pendant cette première partie, que nous appellerons " phase d'acquisition ". La deuxième partie de la boucle principale est aveugle au sens où aucune donnée de capteur ne sera traitée avant la prochaine partie d'acquisition. Nous qualifions cette seconde partie de " phase de commande ". Pendant cette partie aveugle, tous les calculs à partir de l'estimation d'état sont exécutés. Lorsque cela est possible, Paparazzi choisit arbitrairement que chaque partie dure $\frac{T}{2}$.

Au début de la phase d'acquisition, à partir de l'instant t, des demandes sont envoyées aux capteurs maître-sbires qui doivent être lus pendant la période en cours. Leur réponse doit être arrivée avant $t+\frac{T}{2}$. Entre t et $t+\frac{T}{2}$, la boucle acquiert toutes les informations de capteurs requises, y compris l'arrivée sporadique de données, tout en traitant la télémétrie entrante. À $t+\frac{T}{2}$, l'autopilote entre dans sa phase de commande, lance l'estimation d'état, et après cela, exécute chaque fonction d'autopilote selon sa période respective (chaque période de fonction peut être exprimée comme un multiple de T).

Il y a plusieurs avantages à cette architecture : il n'y a pas de problème de cohérence, car chaque fonction dans la partie aveugle a la même perception du système (partage le même état). Si la boucle principale peut être exécutée entièrement dans sa fenêtre de temps T, alors le délai entre l'arrivée des données des capteurs et leur perception est, au maximum, T. En effet, si une donnée arrive à un instant donné, on devra au plus attendre la prochaine phase d'acquisition pour vérifier ces données. Cette boucle interne n'étant pas en concurrence avec d'autres tâches, l'exécution des lectures de capteurs est relativement régulière dans le temps, et s'exécute à peu près régulièrement à chaque période. Ensuite, si les données perçues doivent être traitées par une fonction de période T (par exemple, l'unité de mesure inertielle), elles sont traitées pendant la phase de commande suivant de la même période que leur acquisition, ce qui signifie que le délai maximal entre la réception des données (ou la demande envoyée dans le cas des capteurs maître-sbires) et le traitement des données et l'action sur les actionneurs est de $1,5 \cdot T$. Si la fonction qui traite les données est sporadique, nous observons le même délai. Enfin, si la fonction de traitement est périodique de période $k \cdot T$, alors dans le pire des cas, la fonction en charge du traitement des données entrantes a été exécutée dans la période précédente, et nous devrons attendre la prochaine période de la fonction, $(k-1) \cdot T$ plus tard, pour voir l'exécution du gestionnaire de données périodique. Dans ce cas, le délai maximal entre l'arrivée des données et le traitement suivi de l'action sur les actionneurs est de $(k+\frac{1}{2})T$. Nous n'avons pas mentionné ici le cas des capteurs maître-sbires, car il est évident que si le traitement d'une donnée issue de ce type de capteurs a une période $k \cdot T$, la demande sera envoyée dans la même période que celle où le traitement est exécuté.

Notons que si la durée d'une transaction maître-minion est trop longue par rapport à la période de la boucle principale, elle pourrait être traitée en envoyant la demande pendant une phase d'acquisition, et en obtenant la réponse en fin d'une prochaine phase d'acquisition. Ce serait le cas, par exemple, si T valait 500 microsecondes (période de 2 kHz), et que le délai de réponse d'un capteur I2C était de 400 microsecondes. Avec une phase d'acquisition limitée à 250 microsecondes, il y aurait une période de décalage entre requête et réception de la réponse du capteur.

Dans ce cas, un concepteur pourrait modifier la taille des deux parties pour laisser

plus de temps à la phase de commande, où la plupart des calculs se produisent, ou pour laisser plus de temps à la phase d'acquisition parce que certains capteurs en ont besoin. Nous n'abordons pas cette possibilité dans ce travail. Cependant, la simple implication de la mesure du temps maximal utilisé par la phase de commande permettrait de savoir quelle variation peut être appliquée.

Cette simple architecture d'exécutif statique permet de calculer facilement les délais de bout en bout des chaînes fonctionnelles, des capteurs aux actionneurs, tant que nous pouvons garantir que tous les calculs sont toujours exécutés dans la période T.

5.7 Concepts spécifiques dans l'architecture logicielle des autopilotes

Dans l'optique de montrer à un concepteur d'autopilote personnalisé la structure interne d'un autopilote, ou de le modéliser en vue d'effectuer des analyses, il ne serait pas possible de représenter dans un ADL standard certains concepts clés des autopilotes. Ainsi, une première phase a consisté à identifier les artefacts de modélisation spécifiques qu'il était nécessaire d'introduire dans le but de représenter finement un autopilote.

Le métamodèle basé sur Ecore présenté dans la figure 5.5 présente les éléments essentiels pour modéliser l'architecture logicielle d'un autopilote. Afin de simplifier sa présentation, nous utilisons des concepts AADL (Architecture Analysis and Design Language) [SAEa] tels que les processeurs, les processus, les tâches et les ports. Pour simplifier, nous ne nous référons pas au métamodèle AADL réel, mais les concepts présentés ici ont une correspondance un-à-un immédiate avec les concepts d'implémentation d'AADL qui sont liés aux architectures logicielles multitâche.

Des concepts spécifiques ont été introduits dans notre représentation du logiciel d'autopilote par rapport aux ADL classiques tels qu'AADL, pour prendre en charge les constructions logicielles spécifiques trouvées au cœur des autopilotes. Ils sont liés au concept de Fonction (qui est étroitement lié au concept de sous-programme dans AADL). Ce dernier peut être composé de plusieurs sous-fonctions, qui peuvent à leur tour être composées d'autres sous-fonctions.

À notre connaissance, aucun ADL n'offre nativement la possibilité de représenter les rythmes d'activation des fonctions au sein d'une tâche. La tâche exécutant la boucle principale d'un autopilote contient des fonctions périodiques et sporadiques, ainsi que des transactions périodiques (communications maître-minion).

Le concept de bus logiciel est central dans les systèmes d'autopilote, car il sert de moyen de communication entre les fonctions, en fonction de leur portée. Les bus logiciels varient de simples variables globales non protégées à des middlewares internes complexes tels que AirBorne Ivy (ABI) de Paparazzi, uORB de PX4 ou des middlewares standard comme ROS ou ROS2. Nous les classons ici en fonction de leur portée.

Portée d'une seule tâche Dans le cœur de l'autopilote, Paparazzi utilise des variables globales non protégées pour permettre la communication entre les fonctions. Cette approche augmente la vitesse d'exécution en évitant d'avoir à passer des copies ou des adresses de valeurs en tant que paramètres de fonctions, mais au prix de la lisibilité. Pour

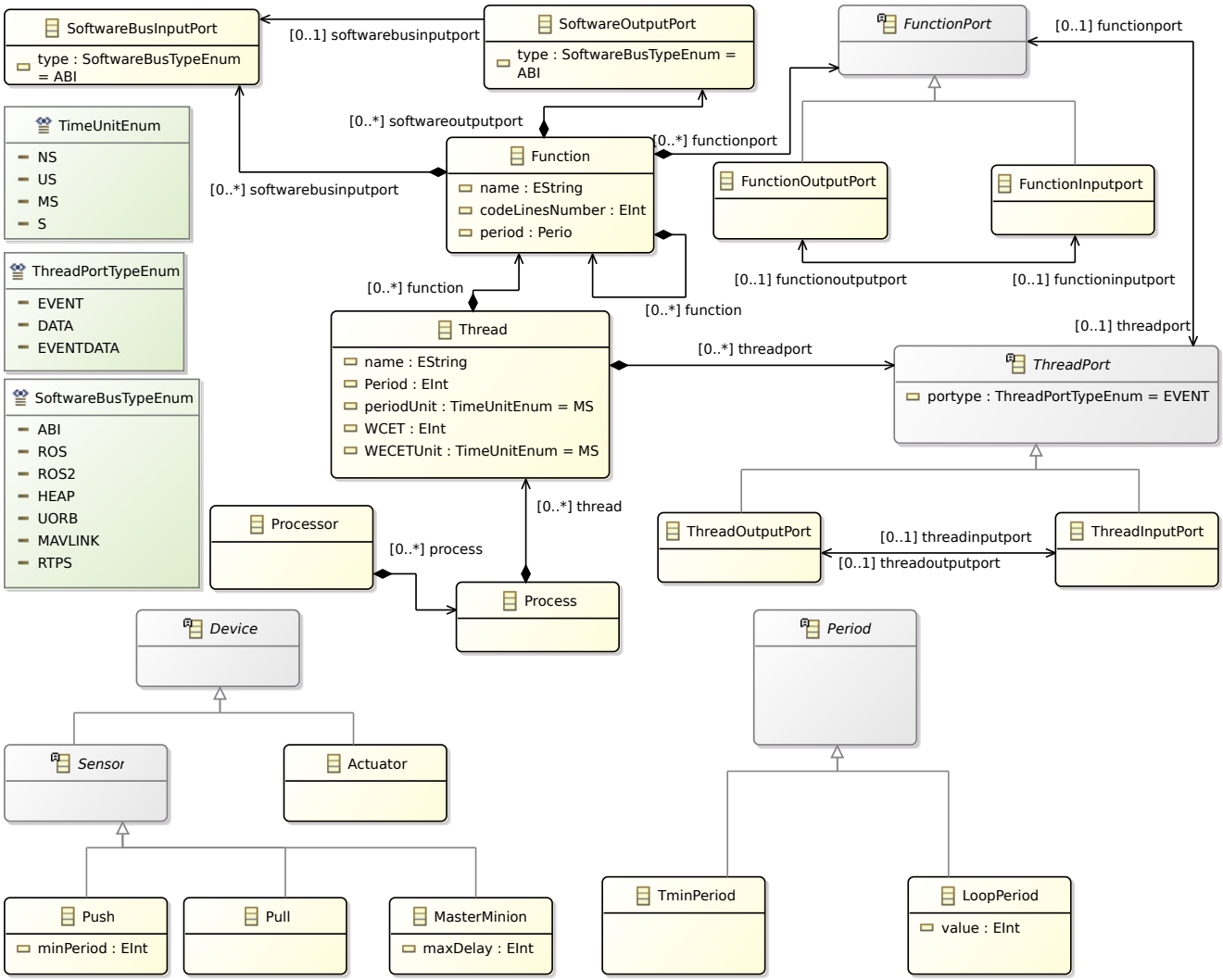

Figure 5.5: Métamodèle de la partie fonctionnelle du point de vue AADL-Like

déterminer quelles variables une fonction lit ou écrit, il faut examiner son code. Paparazzi propose également ABI, un middleware qui abstrait les variables globales en objets middleware. Cependant, pour des raisons d'efficacité, ces méthodes ne sont pas thread-safe car elles reposent sur des variables globales non protégées.

Portée d'un seul processus PX4 utilise uORB, un courtier de requêtes objet, qui prend en charge des mécanismes de publication/abonnement basés sur des sujets (*topics*). Cela ressemble aux mécanismes de ROS et ROS2, où les messages ont une structure de données définie et sont identifiés par un sujet. PX4 comprend une large gamme de sujets qui couvrent la plupart des objets échangés par les fonctions, ce qui améliore la lisibilité et permet à plusieurs tâches d'accéder à ces sujets. Cela se fait au détriment de l'efficacité par rapport à ABI de Paparazzi. D'une façon générale, des bus logiciels personnalisés et thread-safe peuvent être conçus en utilisant les fonctions du système d'exploitation sous-jacent, telles que les mutex POSIX ou les variables conditionnelles.

Portée d'un seul système d'exploitation À notre connaissance, aucun middleware d'autopilote ne fonctionne dans cette portée. Cependant, il est possible d'utiliser l'API du système d'exploitation sous-jacent, telles que les fonctions de communication interprocessus POSIX, pour implémenter un bus logiciel personnalisé ayant cette portée.

Portée d'un seul processeur partitionné Les processeurs peuvent être partagés par plusieurs systèmes d'exploitation partitionnés, tels que ceux qui suivent la norme ARINC653. Bien qu'à notre connaissance, aucun autopilote open source ne fournisse un bus logiciel dans cette portée, il est possible d'utiliser des fonctions de niveau APEX pour implémenter un tel bus.

Portée de plusieurs clusters sur un HMPSoC Sur les unités de traitement multiprocesseurs symétriques et asymétriques, des méthodes de communication standard entre différents clusters ont été proposées, telles que OpenMP et OpenAMP, ce qui facilite l'implémentation d'un bus logiciel à ce niveau. Cependant, à notre connaissance, aucun autopilote open source n'a mis en œuvre un bus logiciel ayant cette portée spécifique.

Portée de différentes plateformes Robot Operating System (ROS), un middleware de robotique open source développé depuis 2007 à l'Université de Stanford et plus tard par Willow Garage jusqu'en 2013, permet à plusieurs nœuds de communiquer sur un réseau. Aujourd'hui, ROS est compatible avec la plupart des autopilotes open source et est un choix souvent retenu par les fabricants de drones pour incorporer des modules personnalisés à l'aide de cartes compagnon ou d'une puissante carte CPU principale. Cependant, des problèmes de performances limitent ce middleware aux fonctions personnalisées relativement lentes, supportant des délais entre estimation d'état et exécution de la fonction personnalisées de l'ordre de dizaines, voire centaines, de millisecondes.

Toutes ces communications middleware peuvent être représentées par leurs chemins complets dans les ADL. Cependant, nous estimons que cette représentation peut être simplifiée par abstraction. Par exemple, si la représentation du chemin complet d'un middleware permettant la communication de fonctions sur différentes plateformes peut

trop compliquer la représentation, l'utilisation de variables globales dans les fonctions peut obscurcir la compréhension de l'impact d'une fonction. C'est pourquoi, dans notre DSL, nous choisissons de représenter chaque bus logiciel par des artefacts de modélisation d'entrée/sortie abstrayant le chemin suivi.

En d'autres termes, il est plus important de comprendre les entrées et les sorties d'une fonction qu'il ne l'est de suivre le chemin complet de la communication middleware.

Par exemple, si une fonction lit une variable globale et écrit une autre variable globale, nous pouvons simplement représenter cette fonction comme une boîte avec deux entrées et deux sorties, sans avoir à représenter le chemin complet de la communication middleware.

Cette approche est plus simple et plus intuitive, et elle permet une meilleure compréhension de l'impact des fonctions sur le système.

Afin de ne pas noyer l'utilisateur sous des milliers de fonctions bas niveau liées soit au système d'exploitation, soit aux pilotes de périphériques, nous désignons quels modules ont un intérêt fonctionnel, et doivent être représentés sous forme de décomposition fonctionnelle, et quels modules sont qualifiés de modules bas niveau ne se voient pas décomposés sous forme de sous-fonctions.

5.8 Implémentation du Framework

L'implémentation du framework proposé s'articule autour de deux étapes principales : l'extraction et la visualisation. L'étape d'extraction vise à extraire les informations pertinentes à partir du code source de l'autopilote à l'aide de techniques d'analyse syntaxique. L'étape de visualisation transforme ces informations extraites en un modèle architectural conforme au métamodèle défini précédemment, dans le but de représenter visuellement la décomposition fonctionnelle et multitâche ainsi que les flots de données.

5.8.1 Extraction

Notre framework est basé sur *ANTLR* [Par22]. Il comprend un moteur d'analyse syntaxique, qui est responsable de la traversée et de la transformation de l'arbre d'analyse. Le traitement se déroule en trois couches, la couche supérieure étant le programme de parcours de l'arbre et de transformation texte en texte. Ce programme est construit sur les deux autres couches fournies par *ANTLR*, qui sont l'arbre d'analyse syntaxique construit et les composants générés (analyseur lexical, analyseur syntaxique, lexèmes et fonctions de traitement).

Pour construire l'arbre d'analyse, lors de la compilation du code source d'une instance d'autopilote générée par Paparazzi, nous ajoutons à GCC les options de compilation permettant de générer, pour chaque fichier C ou C++ compilé, le code GIMPLE [com23b]. Nous avons choisi ce code intermédiaire de représentation en code trois adresses le code généré par GCC, afin de nous abstraire du langage source (C ou C++), et surtout d'avoir uniquement des instructions simples (trois adresses) à traiter, plutôt que des instructions très complexes que l'on pourrait trouver dans le code natif C ou C++. Ce code intermédiaire est conforme à la grammaire GIMPLE (comme illustré dans le listing 5.1). Nous l'analysons à l'aide des trois composants de la première couche, à savoir l'analyseur syntaxique (parseur), l'analyseur lexical (lexer) et les lexèmes (tokens). Une fois que l'arbre

d'analyse syntaxique est construit, il est ensuite transformé en code XML à l'aide des fonctions de traitement générées par la première couche.

Listing 5.1: Extrait de code de la grammaire GILMPLE.

```
...
functionDefinition
        :       attributeSpecifierSeq? declSpecifierSeq?
           declarator virtualSpecifierSeq? functionBody
        |       gimplePreamble? declarator virtualSpecifierSeq?
           functionBody
        ;
gimplePreamble
    : Doublesemi Function idExpression LeftParen (Identifier|
       UnsetAsmName) Comma Identifier Assign IntegerLiteral
       Comma Identifier Assign IntegerLiteral Comma Identifier
       Assign IntegerLiteral Comma Identifier Assign
       IntegerLiteral RightParen
    ;
functionBody:
        constructorInitializer? compoundStatement
        | functionTryBlock
        | Assign (Default | Delete) Semi;
      ...
}
```

5.8.2 Visualisation

Cette étape consiste à visualiser la composition multitâche et fonctionnelle ainsi que les flots de données d'exécution des fonctions en montrant les communications entre elles. Elle prend en entrée l'arbre d'analyse syntaxique généré à la fin de l'étape d'extraction et donne en sortie un modèle qui montre l'ensemble des tâches, l'ensemble des fonctions et leurs sous-fonctions, l'ordre des appels de fonctions et les communications qui se produisent entre elles, en particulier à l'aide de bus logiciels.

La visualisation nécessite une transformation modèle-vers-modèle (M2M) du code, et cela doit être fait tout au long du parcours de l'arbre d'analyse syntaxique. Au fur et à mesure que l'arbre est parcouru, les éléments de l'arbre sont évalués, et ils sont transformés en éléments de modèle équivalents compatibles avec l'outil de visualisation choisi.

L'objectif de la visualisation n'est pas seulement graphique, car le modèle ainsi généré doit permettre de réaliser des analyses, en particulier des exigences non fonctionnelles (comme les délais de bout en bout).

instruction GIMPLE	Concept Capella
pthread_create	AADLThread
pthread_create $3^{ème}$ paramètre	AADLFunction
Appel d'une fonction lors de la définition d'une fonction	sub-function (AADLFunction)
Variable globale accès en écriture	SoftwareBusOutputPort
Variable globale accès en lecture	SoftwareBusInputPort
Variable globale accès en lecture / écriture dans la même fonction	FunctionExchange

Tableau 5.3: Mapping des instruction GIMPLE en concepts Capella et AADL-Like

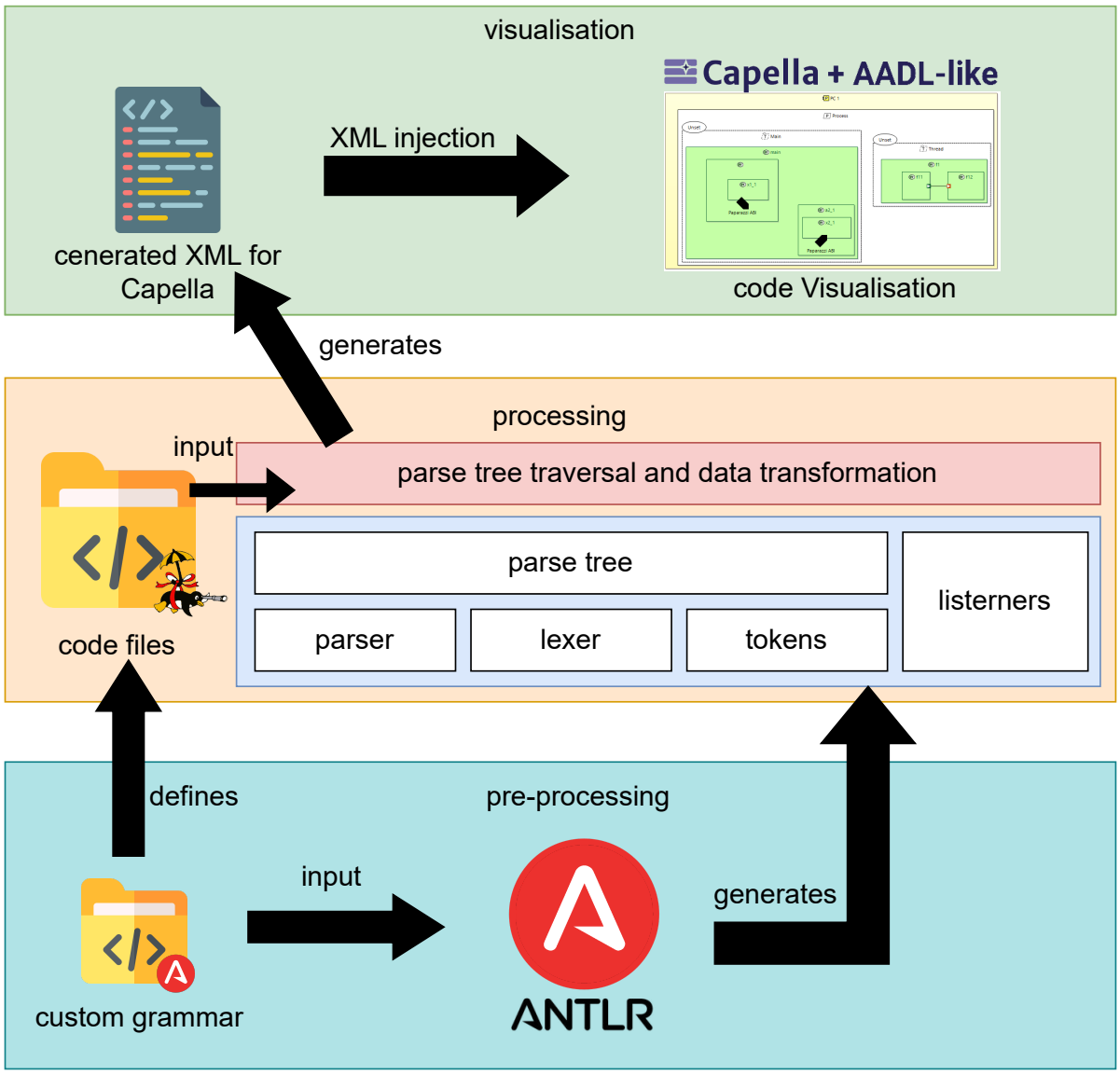

Figure 5.6: Aperçu de la mise en œuvre du framework

La figure 5.7 représente deux parties du point de vue : sur le côté gauche, on peut voir toutes les tâches d'un autopilote Paparazzi basé sur ChibiOS [com23a] : ceux marqués (1) sont quatre tâches en charge de l'UART, à savoir `thd_uart2_tx`, thd_uart7_tx, `thd_uart8_tx` et `thd_uart7_rx` (voir figure 5.8) ayant une fréquence de 100 Hz, donc

$T = 1s$, (2) est un thread shell, (3) est une tâche de journalisation, et (4) est la tâche principale de l'autopilote. Sur le côté droit, (5) est un extrait des fonctions présentes dans la tâche principale de l'autopilote, (6) est un exemple d'une fonction communicant à travers des bus logiciels. Ce sont des fonctions appelées pour vérifier si le UAV a perdu la communication avec la station sol ou est trop haut par rapport à l'altitude autorisée, et dans les deux cas, changeront le mode actuel de l'autopilote. Cette représentation donne une vue globale de l'implémentation de Paparazzi sur la plateforme de Rotorcraft Ulysse, qui nécessite la compilation de 161 fichiers C et C++. Tout en bas du côté gauche, on peut voir des flèches vertes incurvées vers le haut et vers le bas pour chaque accès (lecture ou écriture) à un bus logiciel (variables globales ou middleware). Plusieurs propriétés sont accessibles à partir de l'environnement pour chaque élément représenté, mais dans ce mode, nous avons directement affiché le champ LOC (*lines of codes* pour lignes de code) pour indiquer rapidement au développeur quelles fonctions peuvent être intensives en calcul et nécessitent un calcul du WCET.

5.8.3 Évaluation du framework

Notre chaîne d'outils extrait automatiquement la décomposition multitâche d'un autopilote, comme le montre la figure 5.7, où sept tâches ont été identifiés. À l'heure actuelle, les périodes d'activation des tâches sont dérivées de façon ad-hoc des fichiers de configuration XML de Paparazzi. Certains aspects du flots de données ont également été identifiés, bien que la représentation des blocs de code alternatifs nécessite un affinage supplémentaire. En effet, dans la version actuelle, un bloc conditionnel "si alors sinon" sera représenté par le bloc "alors" suivi du bloc "sinon". Nous avons choisi d'afficher le nombre de lignes de code dans les fonctions, qui pourra servir d'indicateur à l'utilisateur de fonctions potentiellement complexes qui peuvent justifier un examen approfondi, étant donné notre incapacité à calculer automatiquement un temps d'exécution pire cas (cela nécessiterait en effet des mesures dynamiques ou une analyse statique pour la cible). Comme le montre le bas de la figure 5.7, les ports d'entrée et de sortie, nommés globalement d'après les variables globales ou les sujets selon le middleware, sont affichés pour identifier rapidement les dépendances des fonctions et leurs interactions avec les bus logiciels.

En comparaison de la tâche fastidieuse de lire les fichiers source et de tenter de comprendre lesquels sont utilisés, notre représentation vise à assister les constructeurs de drones à comprendre le fonctionnement de leur autopilote, le rôle de chaque fonction et leurs fréquences d'exécution. Cette compréhension simplifie le processus d'ajout d'une fonction personnalisée, soit au cœur de l'autopilote (fonctions représentées) ou en tant que module externe, puisque l'utilisateur peut identifier où et quand les bus logiciels sont lus et écrits. Ceci est illustré dans l'extrait donné dans la figure 5.9. Le modèle peut directement identifier la plupart des constructions d'autopilote telles que les différentes phases du cœur de l'autopilote Paparazzi, les fonctions de télémétrie, la réponse aux états anormaux, ou les lectures de capteurs (par ex., requêtes I2C et lecture de réponses). Notons que dans le cas d'étude considéré, puisqu'il y a un système d'exploitation sous-jacent activant le multi-threading, les capteurs push et maître-sbires sont implémentés en tant que tâches afin de s'appuyer sur les interruptions générées par les données entrantes pour déclencher les lectures de capteurs effectives. Dans ce cas, il y a une communication de type tableau noir [1] entre ces tâches d'entrée/sortie et la partie principale de l'autopilote.

[1] Le tableau noir est un moyen de communication asynchrone : il utilise une zone de mémoire commune

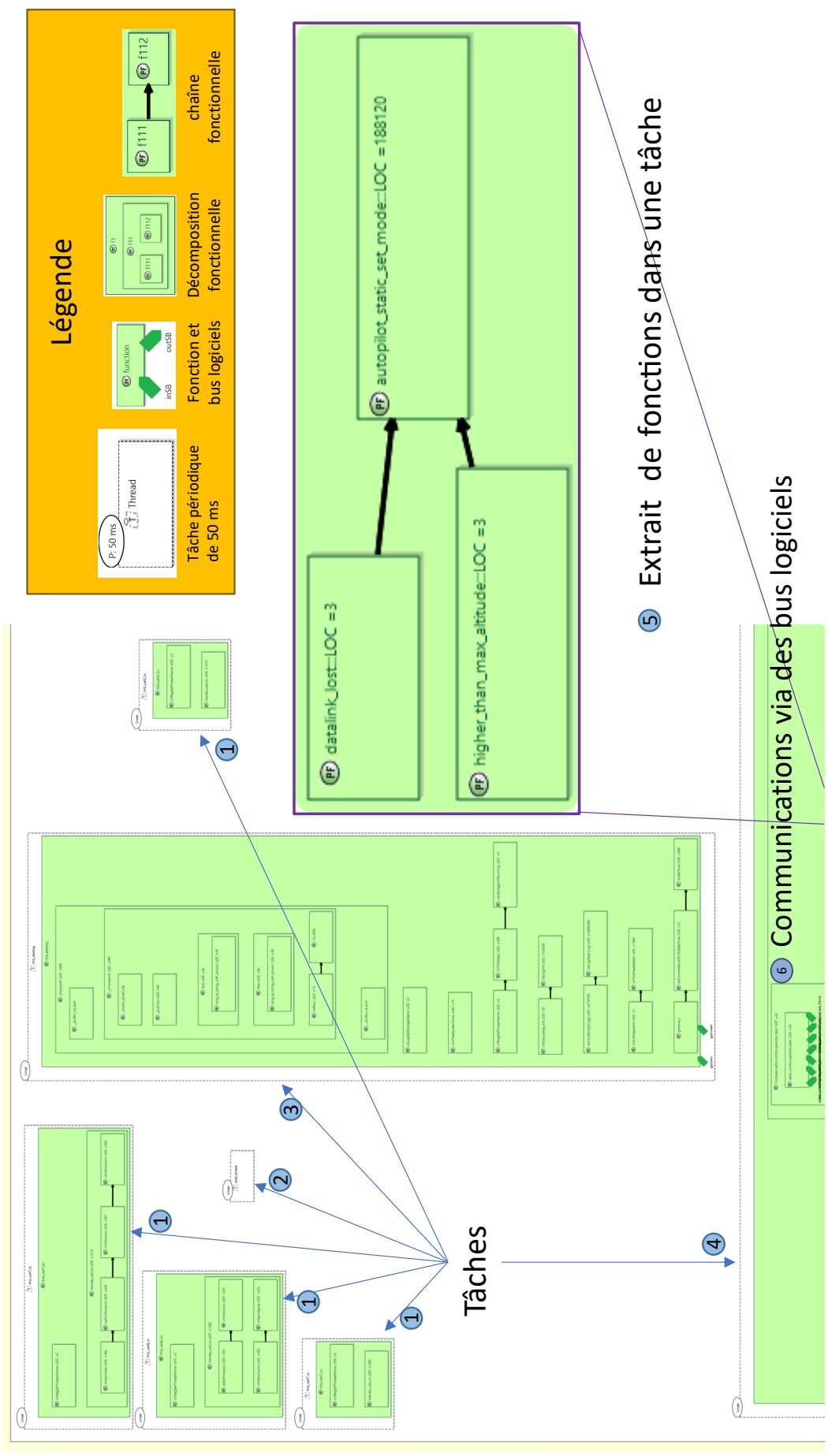

Figure 5.7: Extrait de la représentation de Paparazzi dans le point de vue AADL-like

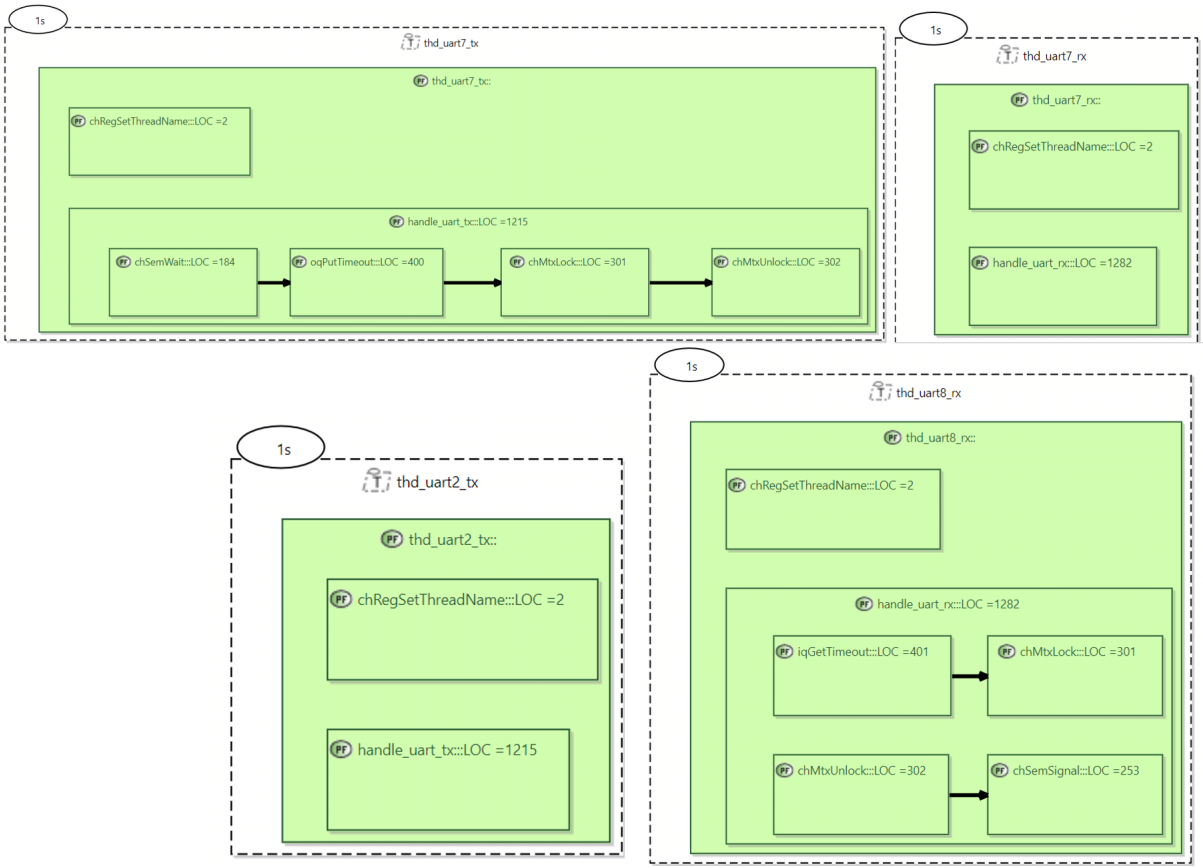

Figure 5.8: Les quatre tâches en charge de l'UART

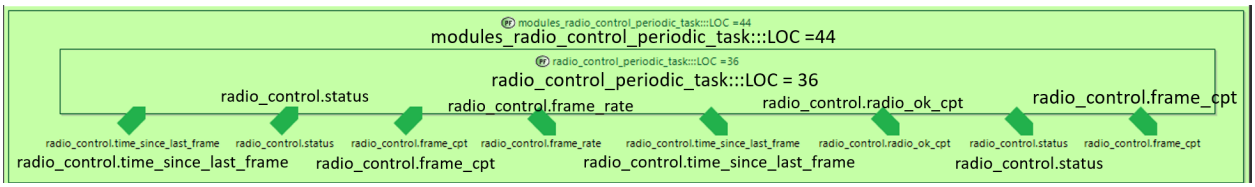

Figure 5.9: La fonction périodique appelée pour la télémétrie utilise plusieurs variables globales non protégées regroupées dans une structure appelée radio_control.

Notre chaîne d'outils fonctionne avec les versions bare-metal, ChibiOS et POSIX de Paparazzi. Bien que conçue initialement pour POSIX, la chaîne d'outils a été étendue à ChibiOS en moins d'une journée, la seule modification nécessaire étant l'ajout des détails de création de tâche spécifiques à ChibiOS. Ainsi, nous pensons que notre framework peut facilement être étendu à n'importe quel drone et n'importe quelle plateforme que Paparazzi peut cibler.

La représentation du modèle actuel reste complexe, et à l'avenir, nous visons à fournir des vues hiérarchiques intermédiaires simplifiées de chaque fonction en utilisant des sous-diagrammes pour afficher les détails internes des fonctions. Le portage vers PX4, qui utilise principalement C++, est encore en cours, mais l'utilisation de code GIMPLE facilite considérablement ce processus.

pouvant contenir un message. L'écriture d'un message écrase le message précédent, et la lecture est non bloquante et non destructive (une valeur déjà lue peut être relue tant qu'elle n'a pas été écrasée par une écriture).

Un autre avantage de notre approche est qu'elle comble le vide entre les fichiers C/C++/Makefile/configuration et les outils d'analyse ou d'optimisation, qui dépendent généralement d'un modèle d'entrée. Pour illustrer cette capacité, dans la suite, nous utilisons un outil pour aider à configurer des modules additionnels de façon à limiter leur impact sur les performances du cœur d'autopilote. Nous supposons qu'un concepteur a décidé d'ajouter des modules au cœur d'autopilote Paparazzi. Grâce au framework que nous proposons, nous supposons qu'il l'a fait en connaissance de ce qui est impacté d'un point de vue fonctionnel. Cependant, nous allons voir qu'il est possible qu'un impact temporel vienne nuire au fonctionnement du cœur d'autopilote, ce que nous pouvons détecter.

5.9 État de l'art sur les systèmes à décalage libre (offset-free)

Lorsque les systèmes industriels constitués de tâches indépendantes sont très chargés, comme dans le domaine automobile, il est fréquent que les décalages des tâches soient positionnés de sorte à lisser la charge. Le problème d'ordonnancement correspondant s'appelle le problème de décalage libre, bien expliqué dans [Goo03]. C'est un problème NP-difficile [Lad+22a]. Par conséquent, des méthodes heuristiques sont largement utilisées pour trouver des décalages permettant d'éviter que plusieurs tâches ne soient en concurrence, ou à défaut limiter les durées dites d'interférence.

Les auteurs de [Goo03] proposent, par exemple, un algorithme qui tente de séparer (décaler) autant que possible les réveils de tâches, paire par paire. Tout d'abord, les paires de tâches sont triées dans un ensemble selon l'ordre décroissant du PGCD (Plus Grand Commun Diviseur) de leurs périodes. Ensuite, chaque paire de tâches se voit attribuer une paire de décalages qui sont espacés de la moitié de leur PGCD (en utilisant un nombre aléatoire comme base pour la paire de décalages).

Une adaptation de ces heuristiques est proposée dans [GGN06] concernant l'ordre dans lequel les paires de tâches sont évaluées : quatre nouveaux principes de tri sont proposés, basés sur des expressions mathématiques contenant les PGCD de chaque paire de tâches. Pour s'accorder avec les notations des travaux cités, la date de réveil initial d'une tâche τ_i sera dans la suite symbolisée par O_i.

De plus, une méthode décrite dans [GHN08] prend la période maximale comme taille d'un intervalle d'analyse et recherche à l'intérieur le plus long intervalle de temps le moins chargé, de sorte qu'un décalage puisse être défini au milieu de celui-ci. Après avoir attribué le décalage à une tâche, ses dates de réveil $O_i + k \cdot T_i$ sont calculées pour remplir l'intervalle d'analyse, de sorte que le plus long intervalle le moins chargé puisse être recalculé pour la tâche suivante, et ainsi de suite.

Paparazzi utilise une technique d'attribution d'un décalage correspondant à un multiple de 10% de chaque période [Pap23] (voir section 5.5). Dans [ASN16], les auteurs utilisent simplement des valeurs générées aléatoirement pour les décalages.

Dans [NDB18], les auteurs proposent d'utiliser un ou plusieurs décalages par tâche pour reproduire, sous FIFO, l'ordonnancement construit par un ordonnanceur optimal (comme le Premier Arrivé, Premier Servi, par exemple). Cependant, la mise en œuvre de

cette méthode nécessiterait la modification, et la revalidation du code de l'autopilote, ce qui est impraticable.

Dans [Lad+22a], les auteurs proposent une méthode appelée *GCD+*, qui est basée sur les propriétés des systèmes à décalage libre plutôt que sur des valeurs arbitraires ou aléatoires.

GCD+ attribue des décalages à des tâches périodiques exécutées sous un ordonnancement FIFO non préemptif. Elle vise à minimiser les interférences et les délais entre les tâches.

L'algorithme commence par calculer le plus grand commun diviseur (PGCD) global de toutes les périodes des tâches, appelé Ω. Ensuite, il divise la période de chaque tâche par Ω pour obtenir sa sous-période T_S. Les tâches sont attribuées à des "sections" en fonction des facteurs premiers de leurs sous-périodes. Les tâches ayant des facteurs premiers partagés dans leurs sous-périodes peuvent partager la même section. Ces sections servent de partitions de l'espace modulaire Ω.

Chaque tâche reçoit un décalage de cycle O_C allant de 0 à $T_S - 1$. Cela la place dans un cycle spécifique de l'espace Ω. Le décalage de section O_S positionne la tâche dans son intervalle de section assigné. Les tâches partageant une section et un cycle sont décalées à l'aide d'un décalage intra-section O_I. Le décalage final O_i est calculé comme $\Omega \cdot O_C + O_S + O_I$.

Les tâches sont assignées aux sections dans l'ordre croissant de la factorisation de leurs sous-périodes pour minimiser les chevauchements. Les tâches avec $T_S = 1$ reçoivent une nouvelle section. Après l'attribution, les tailles des sections sont calculées, et les sections sont décalées de O_S pour éviter les chevauchements. De cette manière, GCD+ aligne les tâches, attribue des décalages échelonnés et prévient les chevauchements et les délais d'interférence.

Selon l'article, cette méthode présente de meilleures performances que d'autres méthodes de l'état de l'art, et elle est donc considérée comme la meilleure approche pour attribuer les décalages de chaque module.

5.10 Application

Pour adapter GCD+ aux besoins de l'exécutif cyclique Paparazzi, il faut prendre en compte plusieurs éléments.

Tout d'abord, il est difficile de déterminer le temps d'exécution exact pire cas C_i de chaque fonction, qui est dans une analyse d'ordonnançabilité considérée comme une tâche non préemptive, car les temps d'exécution sont souvent trop courts pour être mesurés précisément et peuvent dépendre des valeurs d'entrée et de l'environnement physique.

Deuxièmement, l'unité de temps du système est considérée comme étant la période de l'exécutif cyclique (phase d'acquisition suivie de la phase de commande), ce qui signifie que le temps d'exécution des tâches n'est pas une valeur entière, mais une fraction d'une unité de temps. Cependant, GCD+ considère que chaque tâche est périodique, de durée entière inférieure ou égale à la période.

Pour aborder le premier point, nous utilisons le nombre de lignes d'instructions de chaque fonction de code compilé comme une estimation très grossière du temps d'exécution

en pire cas. Cela n'est évidemment pas fiable pour un processus de certification réel, mais se baser sur cette estimation n'affecte pas la généralité de l'approche. De plus, réaliser une véritable analyse du temps d'exécution en pire cas (par exemple, avec des techniques statiques ou dynamiques) de l'ensemble du pilote automatique dépasse le cadre de ce travail.

L'estimation est utilisée pour trier les fonctions par ordre décroissant du C_i estimé. Ce tri est important pour que GCD+ puisse traiter en premier les tâches ayant les temps d'exécution les plus longs, puis les tâches ayant des temps d'exécution plus courts, qui sont placées dans les espaces résiduels. Dans le cas où GCD+ ne peut pas éviter les interférences, cette stratégie peut éviter les interférences entre les tâches longues, en sacrifiant les tâches courtes qui feront l'objet d'interférences.

Ensuite, étant donné que l'unité de temps est la période de l'exécutif cyclique, chaque WCET est considéré inférieur à 1. Ainsi, pour aborder le deuxième point avec GCD+, ils seront considérés comme exactement égaux à 1, de sorte que les modules éviteront d'occuper la même boucle. Nous observons trois configurations intégrant des modules personnalisés dans le cœur d'autopilote de Paparazzi ;

Cas 1 : $n = 11$, $T_i = 10s \ \forall i \leq n$.

- Paparazzi : $O_1 = O_{11} = 0$, $O_i = (i-1)s$.
- GCD+ : $O_i = 2(i-1)\ ms$.

Cas 2 : $T_1 = 6ms$, $T_2 = 60ms$, $T_3 = 66ms$.

- Paparazzi : $O_1 = 0$, $O_2 = 6ms$, $O_3 = 13ms$.
- GCD+ : $O_1 = 0$, $O_2 = 2ms$, $O_3 = 4ms$.

Cas 3 : $T_1 = 1s$, $T_2 = 10s$.

- Paparazzi : $O_1 = 0$, $O_2 = 1s$.
- GCD+ : $O_1 = 0$, $O_2 = 2ms$.

Le tableau 5.4 montre que la stratégie native naïve de Paparazzi ne peut pas éviter les interférences entre les fonctions personnalisées, tandis que GCD+ trouve des décalages adéquats permettant aux fonctions personnalisées de ne jamais se chevaucher. Nous pouvons le constater sur la figure 5.10 : la durée de la partie commande de l'exécution mesurée sur la plateforme cible ne dépasse jamais la milliseconde allouée, et dans le pire des cas, il y a près de 400 μs de marge.

Nous ne traitons ici que trois exemples, mais il est important de rappeler que l'objectif n'est pas d'évaluer GCD+, mais de montrer la faisabilité d'une chaîne d'outils allant du code source aux outils d'analyse et de conception, notamment dans le domaine de l'ordonnancement temps réel. De plus, chaque étude a nécessité une exécution sur une plateforme de drone réelle pour l'évaluation dynamique de performances.

Avec si peu de tâches périodiques personnalisées, notons que nous aurions pu utiliser une méthode exacte (par exemple, basée sur SMT[2]), mais qu'elle ne serait pas évolutive

[2]Satisfiability Modulo Theories fait référence au problème de déterminer si une formule du premier ordre est satisfiable par rapport à une certaine théorie logique.

pour des nombres de modules personnalisés plus importants. Une stratégie pourrait consister à utiliser GCD+ tant que le temps d'exécution mesuré pire cas présente une marge acceptable, et à passer à une méthode optimale dans les cas où GCD+ échoue à le faire.

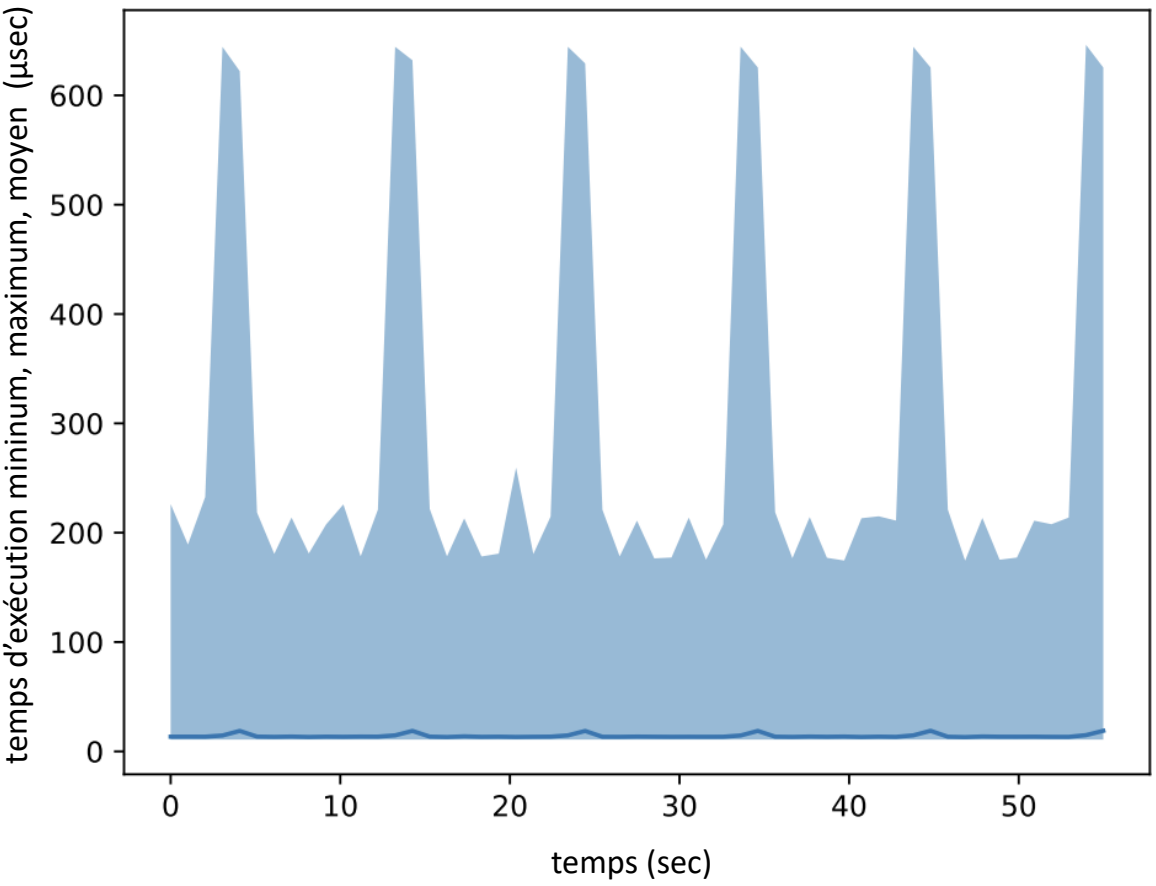

Figure 5.10: Temps d'exécution (moyen, minimum et maximum sur un intervalle de 1 seconde) de la phase de commande de la boucle principale pour le cas 1 en utilisant GCD+.

cas	période (ms)	hyperpériode (s)	interférences par hyperpériode
PPRZ scheduler			
1	1.999	10	1
2	1.998	0.66	10
3	1.998	10	1
GCD+ scheduler			
1	1.999	10	0
2	1.998	0.66	0
3	1.998	10	0

Tableau 5.4: Comparaison du temps d'exécution avec des instants critiques par hyperpériode.

5.11 Conclusion

Dans ce chapitre, nous avons présenté une approche pour extraire, modéliser et analyser l'architecture interne des autopilotes open-source de drones. Ces autopilotes sont des systèmes embarqués temps-réel complexes qu'il est difficile de comprendre et d'étendre uniquement à partir de leur code source.

Nous avons tout d'abord identifié les concepts nécessaires pour représenter les spécificités des autopilotes par rapport aux ADL classiques. Puis, nous avons décrit le framework développé pour extraire automatiquement un modèle de l'architecture logicielle et matérielle à partir du code source. Ce framework repose sur l'analyse syntaxique du code et sa transformation en un modèle conforme à notre métamodèle.

L'approche a été validée sur l'autopilote open-source Paparazzi. Le modèle extrait permet de représenter visuellement la décomposition fonctionnelle et multitâche, ainsi que les flots de données entre les fonctions. Cette représentation graphique facilite la compréhension de l'architecture interne et des interactions entre les différents composants logiciels de l'autopilote.

Nous avons également montré comment le modèle extrait peut être utilisé comme entrée pour des outils d'analyse et de conception. Nous avons adapté et appliqué une technique d'ordonnancement basée sur les systèmes à décalage libre pour intégrer des modules personnalisés.

Cette approche ouvre la voie à l'analyse automatisée des autopilotes open-source et à leur adaptation sûre et fiable grâce à l'extraction de modèles architecturaux. Elle peut être étendue à d'autres types de systèmes embarqués basés sur du code open source. Les travaux futurs viseront à affiner le framework d'extraction et à le connecter à plus d'outils d'analyse pour répondre aux besoins des développeurs de drones.

CHAPITRE 6

Conclusion générale

Cette thèse avait pour ambition d'apporter des réponses aux défis posés par le déploiement des systèmes temps réel critiques, dans un contexte où les informations temporelles sur le comportement des fonctions restent limitées aux phases amont du cycle de développement.

Dans un premier temps, nous avons présenté de manière approfondie le contexte des systèmes embarqués temps réel, en détaillant leurs spécificités, leurs cycles de vie logiciel, et les approches méthodologiques majeures telles qu'AUTOSAR et ARCADIA. Puis, nous avons introduit de façon succincte le domaine de l'ordonnancement temps réel, en exposant les modèles de tâches, les algorithmes d'ordonnancement sur mono et multiprocesseur, ainsi que les tests d'ordonnançabilité permettant de vérifier le respect des contraintes temporelles.

Sur ces bases, nous avons proposé deux contributions originales pour faciliter et optimiser la phase cruciale de déploiement dans le contexte considéré.

La première est la méthode RYM, une technique simple, applicable à la main, de prédéploiement des systèmes critiques monoprocesseur fondée sur les rythmes d'activation. Nous avons montré qu'elle permet de générer une configuration de tâches correcte par construction, en réduisant le nombre de tâches créées ainsi que les temps de réponse sur les chaînes fonctionnelles critiques. Un prototype logiciel a été développé pour valider l'applicabilité de la méthode.

La seconde contribution est un framework original permettant d'extraire à partir du code source C/C++ d'un autopilote un modèle opérationnel et fonctionnel. Ce modèle permet de représenter l'architecture interne d'un autopilote utilisé en tant que Modifiable Off-The-Shelf, et d'analyser l'architecture logicielle interne des autopilotes open-source de drones à partir de leur code source. Ce framework comble le vide entre le code et les outils d'analyse, et a été validé par l'adaptation et l'évaluation d'une technique avancée d'ordonnancement utilisant le décalage d'offset afin d'éviter les interférences temporelles sur l'autopilote Paparazzi.

Ces deux contributions, à la fois méthodologiques et technologiques, fournissent des réponses au problème difficile du déploiement des systèmes temps réel critiques avec des informations partielles.

Concernant la méthode RYM, les perspectives possibles incluent son extension aux architectures multi-cœurs, l'intégration d'objectifs extra-fonctionnels additionnels dans l'optimisation, et une validation expérimentale approfondie sur un large ensemble de cas d'étude.

Pour le framework d'extraction d'architecture des autopilotes, les modèles générés pourraient être exploités par un écosystème étendu d'outils d'analyse, couvrant par exemple l'évaluation de la consommation énergétique ou la vérification formelle. L'automatisation de l'extraction pourrait être améliorée par l'apprentissage automatique, et le passage à l'échelle faisant l'objet d'une étude sur des autopilotes plus volumineux.

De façon plus large, ces travaux démontrent le fort potentiel des techniques de modélisation et d'optimisation pour appréhender la complexité croissante des systèmes embarqués critiques. Ils ouvrent des perspectives pour de futurs travaux visant à transformer la maîtrise de la phase cruciale de déploiement dans l'ingénierie des systèmes autonomes.

Bibliographie

[AS-05] AS-2 Embedded Computing Systems Committee SAE. *Architecture analysis & design language (AADL)*. 2005.

[ASN16] Sebastian Altmeyer, Sakthivel Manikandan Sundharam, and Nicolas Navet. *The case for FIFO real-time scheduling*. Tech. rep. University of Luxembourg, 2016.

[ATE10] ATESST2 Consortium. "Methodology Guidelines When Using EAST-ADL2". In: 2010.

[AK03] C. Atkinson and T. K"uhne. "Model-driven development: a metamodeling foundation". In: *IEEE software* 20.5 (2003), pp. 36–41.

[Aud+93] N. Audsley et al. "Applying new scheduling theory to static priority preemptive scheduling". In: *Software Engineering Journal* 8.5 (1993), pp. 284–292.

[Aud91] Neil C Audsley. "Optimal priority assignment and feasibility of static priority tasks with arbitrary start times". In: (1991).

[Aud+91] Neil C Audsley et al. "Hard real-time scheduling: The deadline-monotonic approach". In: *IFAC Proceedings Volumes* 24.2 (1991), pp. 127–132.

[AUT22] AUTOSAR. "AUTOSAR Methodology". In: (2022).

[AUT23a] AUTOSAR. *AUTOSAR Introduction, Part 1 - The AUTOSAR Partnership and Standardization*. https://www.autosar.org/fileadmin/user_upload/AUTOSAR_Introduction_PDF/AUTOSAR_EXP_Introduction_Part1.pdf. [Online; accessed 24-July-2023]. 2023.

[AUT23b] AUTOSAR. *AUTOSAR Layered Software Architecture*. https://www.autosar.org/fileadmin/standards/R20-11/CP/AUTOSAR_EXP_VFB.pdf. [Online; accessed 25-July-2023]. 2023.

[aut23] autosar. *The standardized software framework for intelligent mobility*. 2023. URL: https://www.autosar.org/ (visited on 02/12/2023).

[AUT11] AUTOSAR Consortium. *AUTOSAR Methodology*. 2011.

[BG01] J. B'ezivin and O. Gerb'e. "Towards a precise definition of the OMG/MDA framework". In: *Proceedings 16th Annual International Conference on Automated Software Engineering (ASE 2001)*. IEEE. 2001, pp. 273–280.

[Bak91] Theodore P. Baker. "Stack-based scheduling of realtime processes". In: *Real-Time Systems* 3.1 (1991), pp. 67–99.

[BLD05] C Bartolini, G Lipari, and M Di Natale. "From functional blocks to the synthesis of the architectural model in embedded real-time applications". In: *RTAS 2005: 11th IEEE Real Time and Embedded Technology and Applications Symposium, Proceedings*. 2005.

[Bar+90] S. K. Baruah et al. "Preemptively scheduling hard-real-time sporadic tasks on one processor". In: *IEEE Real-Time Systems Symposium*. 1990, pp. 182–190.

[Bar98a] S. K. Baruah. "A general model for recurring real-time tasks". In: *IEEE Real-Time Systems Symposium*. 1998, pp. 114–122.

[Bar98b] S. K. Baruah. "Feasibility Analysis of Recurring Branching Tasks". In: *Euromicro Workshop on Real-Time Systems*. 1998, pp. 138–145.

[Bar+99] S. K. Baruah et al. “Generalized multiframe tasks”. In: *Real-Time Systems* 17.1 (1999), pp. 5–22.

[Bar10] S. K. Baruah. “The Non-cyclic Recurring Real-Time Task Model”. In: *IEEE Real-Time Systems Symposium*. 2010, pp. 173–182.

[BB06] Sanjoy Baruah and Alan Burns. “Sustainable scheduling analysis”. In: *2006 27th IEEE International Real-Time Systems Symposium (RTSS'06)*. IEEE. 2006, pp. 159–168.

[Bar03] Sanjoy K Baruah. “Dynamic-and static-priority scheduling of recurring real-time tasks”. In: *Real-Time Systems* 24 (2003), pp. 93–128.

[BMR90] Sanjoy K Baruah, Aloysius K Mok, and Louis E Rosier. “Preemptively scheduling hard-real-time sporadic tasks on one processor”. In: *[1990] Proceedings 11th Real-Time Systems Symposium*. IEEE. 1990, pp. 182–190.

[Bar+93] Sanjoy K Baruah et al. “Proportionate progress: A notion of fairness in resource allocation”. In: *Proceedings of the twenty-fifth annual ACM symposium on Theory of computing*. 1993, pp. 345–354.

[BFO14] Antoine Bertout, Julien Forget, and Richard Olejnik. “Minimizing a real-time task set through task clustering”. In: *Proceedings of the 22nd International Conference on Real-Time Networks and Systems*. ACM. 2014, p. 23.

[BK11] Dirk Beyer and M Erkan Keremoglu. “CPAchecker: A tool for configurable software verification”. In: *International Conference on Computer Aided Verification* (2011), pp. 184–190.

[Bou+16] Rahma Bouaziz et al. “Architecture Exploration of Real-Time Systems Based on Multi-objective Optimization”. In: *Proceedings of the IEEE International Conference on Engineering of Complex Computer Systems, ICECCS* 2016-Janua (2016), pp. 1–10.

[Bou+18] Rahma Bouaziz et al. “Multi-objective design exploration approach for Ravenscar real-time systems”. In: *Real Time Syst.* 54.2 (2018), pp. 424–483. DOI: 10.1007/s11241-018-9299-6. URL: https://doi.org/10.1007/s11241-018-9299-6.

[Bra+14] T. Braun et al. “Model-based Extension of AUTOSAR for Architectural Virtual Prototyping”. In: *SAE 2014 World Congress & Exhibition*. 2014.

[Bro11] Benjamin Brosgol. “Do-178c: the next avionics safety standard”. In: *ACM SIGAda Ada Letters* 31.3 (2011), pp. 5–6.

[Bro04] Alan W Brown. “Model driven architecture: Principles and practice”. In: *Software and systems modeling* 3.4 (2004), pp. 314–327.

[Bur99] Alan Burns. “The ravenscar profile”. In: *ACM SIGAda Ada Letters* 19.4 (1999), pp. 49–52.

[But11] Giorgio C. Buttazzo. “Hard Real-Time Computing Systems: Predictable Scheduling Algorithms and Applications, Third Edition”. In: *Real-Time Systems Series*. 2011.

[CDC11] Gerardo Canfora, Massimiliano Di Penta, and Luigi Cerulo. “Achievements and challenges in software reverse engineering”. In: *Communications of the ACM* 54.4 (2011), pp. 142–151.

[Car+04] John Carpenter et al. *A Categorization of Real-Time Multiprocessor Scheduling Problems and Algorithms*. 2004.

[CC89] H Chetto and Maryline Chetto. "End-to-end scheduling to meet deadlines in distributed systems". In: *IEEE Transactions on Computers* 38.9 (1989), pp. 1259–1272.

[com23a] ChibiOS community. *ChibiOS/RT Documentation*. https://www.chibios.org/dokuwiki/doku.php. Accessed: 2023-10-12. 2023.

[com23b] GCC community. *GIMPLE - GNU Compiler Collection (GCC) Internals*. https://gcc.gnu.org/onlinedocs/gccint/GIMPLE.html. Accessed: 2023-10-12. 2023.

[CG14] Francis Cottet and Emmanuel Grolleau. *Systèmes temps réel embarqués-2e éd.-Spécification, conception, implémentation et validation tem: Systèmes temps réel embarqués*. Dunod, 2014.

[Cue+08a] P Cuenot et al. "Developing Automotive Products Using the EAST-ADL2, an AUTOSAR Compliant Architecture Description Language". In: *Ing'enieurs de l'Automobile*. 793. 2008, p. 58.

[Cue+08b] P. Cuenot et al. "Complementarity of EAST-ADL and AUTOSAR for Automotive Architecture Design". In: *ERTS 2008*. 2008.

[Cue+08c] Philippe Cuenot et al. "Developing automotive products using the east-adl2, an autosar compliant architecture description language". In: *European Congress on Embedded Real-Time Software*. 2008.

[Cuo+12] Pascal Cuoq et al. "Frama-C: A software analysis perspective". In: *International Conference on Software Engineering and Formal Methods*. Springer. 2012, pp. 233–247.

[DSo01] D. D'Souza. *Model-Driven Architecture and Integration: Opportunities and Challenges*. Tech. rep. Catalysis Group, 2001.

[Dav14] Robert I. Davis. "A Review of Fixed Priority and EDF Scheduling for Hard Real-Time Uniprocessor Systems". In: *SIGBED Rev.* 11.1 (Feb. 2014), pp. 8–19. DOI: 10.1145/2597457.2597458. URL: https://doi.org/10.1145/2597457.2597458.

[Der74] Michael L Dertouzos. "Control robotics: The procedural control of physical processes". In: *Proceedings IF IP Congress, 1974*. 1974.

[DM89] Michael L. Dertouzos and Aloysius K. Mok. "Multiprocessor online scheduling of hard-real-time tasks". In: *IEEE Transactions on software engineering* 15.12 (1989), pp. 1497–1506.

[EAS18] EAST-ADL Association. "EAST-ADL Domain Model Specification". In: (2018).

[Ecl23] Inc. Eclipse Foundation. *Eclipse*. https://www.eclipse.org/. Accessed: 2023-10-19. 2023.

[EIA03] EIA32. *EIA-632:2003 Processes for Engineering a System*. EIA, 2003.

[ER08] Friedrich Eisenbrand and Thomas Rothvoß. "Static-priority real-time scheduling: Response time computation is NP-hard". In: *2008 Real-Time Systems Symposium*. IEEE. 2008, pp. 397–406.

[FGH06] Peter H Feiler, David P Gluch, and John J Hudak. "The architecture analysis & design language (AADL): An introduction". In: *Technical Report CMU/SEI-2006-TN-011*. 2006.

[Fei+04] Peter H Feiler et al. "Embedded system architecture using SAE AADL". In: *Proceedings of the 2004 Embedded Real Time Software and Systems Conference. Toulouse, France*. 2004.

[Fou23] Eclipse Foundation. *Eclipse Modeling Framework (EMF)*. 2023. URL: https://www.eclipse.org/modeling/emf/ (visited on 08/01/2023).

[GS08] S'ebastien G'erard and Bran Selic. "The UML–MARTE Standardized Profile". In: *17th IFAC World Congress (IFAC'08)* (2008), pp. 6909–6913.

[Gai+07] Paolo Gai et al. *Adding timing analysis to functional design to predict implementation errors*. Tech. rep. SAE Technical Paper, 2007.

[Gmba] Vector Informatik GmbH. *VectorCAST*. https://www.vector.com/.

[Gmbb] Zoom GmbH. *Zoom*. https://www.rotateright.com/zoom/.

[Goo03] Joël Goossens. "Scheduling of offset free systems". In: *Real-Time Systems* 24.2 (2003), pp. 239–258.

[GMS94] Michel Goossens, Frank Mittelbach, and Alexander Samarin. *The LaTeX Companion*. Reading, Massachusetts: Addison-Wesley, 1994.

[Gra] Nicolas Granjoux. *Shinyprofiler*. https://github.com/nico/Shinyprofiler.

[GGN06] Mathieu Grenier, Joël Goossens, and Nicolas Navet. "Near-optimal fixed priority preemptive scheduling of offset free systems". In: *14th International Conference on Real-Time and Networks Systems (RTNS'06)*. 2006, pp. 35–42.

[GHN08] Mathieu Grenier, Lionel Havet, and Nicolas Navet. "Pushing the Limits of CAN - Scheduling Frames with Offsets Provides a Major Performance Boost". In: *4th European Congress on Embedded Real Time Software (ERTS 2008)* (2008).

[GRO+13] E. GROLLEAU et al. *Ordonnancement temps réel - Ordonnancement monoprocesseur*. https://www.techniques-ingenieur.fr/base-documentaire/technologies-de-l-information-th9/systemes-embarques-42588210/ordonnancement-temps-reel-s8055/architectures-des-applications-temps-reel-s8055v2niv10001.html. [Online; accessed 24-July-2023]. 2013.

[Gro14] Object Management Group. "Model Driven Architecture". In: (2014).

[Gro23a] OSEK/VDX Group. *OSEK/VDX*. http://www.osek-vdx.org/. Accessed: 2023-10-19. 2023.

[Gro23b] Thales Group. *Thales*. https://www.thalesgroup.com/en. Accessed: 2023-10-19. 2023.

[Hau+06] Matthew Hause et al. "The SysML modelling language". In: *Fifteenth European Systems Engineering Conference*. Vol. 9. 2006, pp. 1–12.

[HL92] Kwang Soo Hong and JY-T Leung. "On-line scheduling of real-time tasks". In: *IEEE transactions on Computers* 41.10 (1992), pp. 1326–1331.

[Hor74] WA Horn. "Some simple scheduling algorithms". In: *Naval Research Logistics Quarterly* 21.1 (1974), pp. 177–185.

[HF07] John Hudak and Peter Feiler. *Developing AADL models for control systems: A practitioner's guide*. Tech. rep. Carnegie-Mellon Univ Pittsburgh PA Software Engineering Inst, 2007.

[IEE08] IEEE07. *IEEE 12207-2008 - Systems and software engineering—Software life cycle processes*. IEEE, 2008.

[Inc22] Synopsys Inc. *Synopsys*. https://www.synopsys.com. 2022.

[ISO08a] ISO07. *ISO/IEC 12207:2008 Systems and software engineering - Software life cycle processes*. ISO, 2008.

[ISO08b] ISO88. *ISO/IEC 15288:2008 Systems and software engineering - System life cycle processes*. ISO, 2008.

[JZ16] Konrad Jamrozik and Andreas Zeller. "Oovaide: Weaving dependencies between C++ source files". In: *2016 31st IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE. 2016, pp. 794–799.

[JL11] Andreas Johnsen and Kristina Lundqvist. "Developing dependable software-intensive systems: AADL vs. EAST-ADL". In: *International Conference on Reliable Software Technologies*. Springer. 2011, pp. 103–117.

[JP86] Mathai Joseph and Paritosh Pandya. "Finding response times in a real-time system". In: *The Computer Journal* 29.5 (1986), pp. 390–395.

[Kam+20] Soulimane Kamni et al. "Towards a model-based multi-objective optimization approach for safety-critical real-time systems". In: *2020 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE. 2020, pp. 634–637.

[Kam+22] Soulimane Kamni et al. "A Reverse Design Framework for Modifiable-off-the-Shelf Embedded Systems: Application to Open-Source Autopilots". In: *International Conference on Model and Data Engineering*. Springer. 2022, pp. 133–146.

[Kam+23] Soulimane Kamni et al. "Easing the tuning of drone autopilots through a model-based framework". In: *Journal of Computer Languages* (2023), p. 101240. ISSN: 2590-1184. DOI: https://doi.org/10.1016/j.cola.2023.101240. URL: https://www.sciencedirect.com/science/article/pii/S2590118423000503.

[KWB03] A. Kleppe, J. Warmer, and W. Bast. *MDA explained: the model driven architecture: practice and promise*. Addison-Wesley Professional, 2003.

[Knu86] Donald Knuth. *The TEXbook*. Reading, Massachusetts: Addison-Wesley, 1986.

[KWS03] Sharath Kodase, Shige Wang, and Kang G Shin. "Transforming structural model to runtime model of embedded software with real-time constraints". In: *2003 Design, Automation and Test in Europe Conference and Exhibition*. IEEE. 2003, pp. 170–175.

[Kos03] Rainer Koschke. "Software visualization in software maintenance, reverse engineering, and re-engineering: a research survey". In: *Journal of Software Maintenance and Evolution: Research and Practice* 15.2 (2003), pp. 87–109.

[Kou+19] Anis Koubâa et al. "Micro air vehicle link (mavlink) in a nutshell: A survey". In: *IEEE Access* 7 (2019), pp. 87658–87680.

[Lad+22a] Matheus Ladeira et al. “Scheduling Offset-Free Systems Under FIFO Priority Protocol”. In: *34th Euromicro Conference on Real-Time Systems (ECRTS 2022)*. Vol. 231. 2022.

[Lad+22b] Matheus Ladeira et al. “Scheduling Offset-Free Systems Under FIFO Priority Protocol”. In: *34th Euromicro Conference on Real-Time Systems, ECRTS 2022, July 5-8, 2022, Modena, Italy*. Ed. by Martina Maggio. Vol. 231. LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2022, 11:1–11:19. DOI: 10.4230/LIPIcs.ECRTS.2022.11. URL: https://doi.org/10.4230/LIPIcs.ECRTS.2022.11.

[Lak+19] Wafa Lakhdhar et al. “Multiobjective Optimization Approach for a Portable Development of Reconfigurable Real-Time Systems: From Specification to Implementation”. In: *IEEE Transactions on Systems, Man, and Cybernetics: Systems* 49.3 (2019), pp. 623–637. ISSN: 21682232. DOI: 10.1109/TSMC.2017.2781460.

[Lam94] Leslie Lamport. *LaTeX — A Document Preparation System*. Second. Reading, Massachusetts: Addison-Wesley, 1994.

[LD03] Michele Lanza and Stéphane Ducasse. “Polymetric views-a lightweight visual approach to reverse engineering”. In: *IEEE Transactions on Software Engineering* 29.9 (2003), pp. 782–795.

[LHB18] B. Le Nabec, B. B. Hedia, and J.-P. Babau. “QuaRTOS-DSE: A Tool for Design Space Exploration of Embedded Real-Time System”. In: *2018 IEEE 21st International Symposium on Real-Time Distributed Computing (ISORC)*. 2018, pp. 42–50.

[Leh90] John P Lehoczky. “Fixed priority scheduling of periodic task sets with arbitrary deadlines”. In: *Proceedings of the 11th Real-Time Systems Symposium*. 1990, pp. 201–209.

[LM80] Joseph Y-T Leung and ML Merrill. “A note on preemptive scheduling of periodic, real-time tasks”. In: *Information processing letters* 11.3 (1980), pp. 115–118.

[LW82] Joseph Y-T Leung and Jennifer Whitehead. “On the complexity of fixed-priority scheduling of periodic, real-time tasks”. In: *Performance evaluation* 2.4 (1982), pp. 237–250.

[Lim+09] Didier Lime et al. “Romeo: A parametric model-checker for Petri nets with stopwatches”. In: *Tools and Algorithms for the Construction and Analysis of Systems: 15th International Conference, TACAS 2009, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2009, York, UK, March 22-29, 2009. Proceedings 15*. Springer. 2009, pp. 54–57.

[LL73a] C. L. Liu and J. W. Layland. “Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment”. In: *Journal of the ACM* 20.1 (1973), pp. 46–61.

[LL73b] Chung Laung Liu and James W Layland. “Scheduling algorithms for multiprogramming in a hard-real-time environment”. In: *Journal of the ACM (JACM)* 20.1 (1973), pp. 46–61.

[LLC] PVS-Studio LLC. *PVS-Studio*. https://pvs-studio.com/en/.

[MS10] M. Marouf and Y. Sorel. "Schedulability conditions for non-concrete periodic tasks with precedence constraints". In: *19th International Conference on Real-Time and Network Systems*. 2010, pp. 101–109.

[MAS] MAST. *Modeling and Analysis Suite for Real-Time Applications*. URL: https://mast.unican.es/ (visited on 02/19/2021).

[Meh+13] A. Mehiaoui et al. "A two-step optimization technique for functions placement, partitioning, and priority assignment in distributed systems". In: *ACM SIGPLAN Notices* 48 (2013). ISSN: 03621340. DOI: 10.1145/2499369.2465572.

[Meh+18] Asma Mehiaoui et al. "Optimizing the deployment of tree-shaped functional graphs of real-time system on distributed architectures". In: *Automated Software Engineering* (2018), pp. 1–57. ISSN: 15737535. DOI: 10.1007/s10515-018-0244-7. URL: https://doi.org/10.1007/s10515-018-0244-7.

[MIL94] MILD98. *MIL-STD-498:1994 U.S. Department of Defense Standard - Software development and documentation*. U.S. DoD, 1994.

[MOF15] OMG MOF. *OMG Meta Object Facility (MOF) Core Specification. Version 2.4. 2. April 2014*. 2015. URL: http://www.omg.org/spec/MOF/2.4.1/.

[Moh15] SP Mohanty. "Metamodel-based fast AMS-SoC design methodologies". In: *Nanoelectronic Mixed-Signal System Design*. McGraw-Hill, 2015.

[Mok83a] A. K. Mok. "Fundamental Design Problems of Distributed Systems for The Hard-Real-Time Environment". In: *PhD thesis, Massachusetts Institute of Technology*. 1983.

[MC97] A. K. Mok and D. Chen. "A Multiframe Model for Real-Time Tasks". In: *IEEE Transactions on Software Engineering* 23.10 (1997), pp. 635–645.

[Mok83b] Aloysius Ka-Lau Mok. "Fundamental design problems of distributed systems for the hard-real-time environment". PhD thesis. Massachusetts Institute of Technology, 1983.

[Moy+10] N. T. Moyo et al. "On Schedulability Analysis of Non-cyclic Generalized Multiframe Tasks". In: *Euromicro Conference on Real-Time Systems*. 2010, pp. 271–278.

[Nam+09] Min-Young Nam et al. "Asiist: Application specific i/o integration support tool for real-time bus architecture designs". In: *2009 14th IEEE International Conference on Engineering of Complex Computer Systems*. IEEE. 2009, pp. 11–22.

[NDB18] Mitra Nasri, Robert I Davis, and Björn B Brandenburg. "FIFO with offsets: High schedulability with low overheads". In: *2018 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*. IEEE. 2018, pp. 271–282.

[nat23] nato. *NATO Architecture Framework, Version 4*. https://www.nato.int/cps/en/natohq/topics_157575.htm. Accessed: 2023-08-01. 2023.

[NE04] V Normand and D Exertier. "Model-driven systems engineering: SysML & the MDSysE approach at Thales". In: *Ecole d'été CEA-ENSIETA, Brest, France* 38 (2004), p. 50.

[Obj23] Object Management Group. *Model Driven Architecture Home Page*. http://www.omg.org/mda/. 2023.

[OMG15] OMG. *XML Metadata Interchange (XMI) Specification*. 2015. URL: https://www.omg.org/spec/XMI/2.5.1.

[OMG23] OMG. *OMG MARTE*. https://www.omg.org/omgmarte/. Accessed: 2023-09-05. 2023.

[OMG01] OMG Architecture Board MDA Drafting Team. *Model-Driven Architecture: A Technical Perspective*. Tech. rep. Object Management Group, 2001.

[OSA23] OSATE. *OSATE - Open Source AADL Tool Environment*. Web page. Accessed: 2023-03-09. 2023. URL: https://osate.org/.

[PH98] José Carlos Palencia and M González Harbour. "Schedulability analysis for tasks with static and dynamic offsets". In: *Proceedings 19th IEEE Real-Time Systems Symposium (Cat. No. 98CB36279)*. IEEE. 1998, pp. 26–37.

[Pap23] Paparazzi developers. *Paparazzi offset generation source code*. https://github.com/paparazzi/paparazzi/blob/master/sw/tools/generators/gen_periodic.ml. Accessed: 03-08-2023. 2023.

[Par22] Terence Parr. *ANTLR (ANother Tool for Language Recognition)*. Accessed: January 16, 2024. 2022. URL: https://www.antlr.org/.

[Pol23] Polarsys. *Capella: OPEN SOURCE SOLUTION FOR MODEL-BASED SYSTEMS ENGINEERING*. https://www.polarsys.org/capella/. Accessed: 2022-08-01. 2023.

[Pos80] Jon Postel. *User datagram protocol*. Tech. rep. 1980.

[Qui09] Morgan Quigley. "ROS: an open-source Robot Operating System". In: *IEEE International Conference on Robotics and Automation*. 2009.

[Qur+10] TN Qureshi et al. "From EAST-ADL to AUTOSAR Software Architecture: A Mapping Scheme". In: *3rd Workshop on Model-Driven Tool & Process Integration*. 2010.

[Rah+12] Ahmed Rahni et al. "Feasibility analysis of real-time transactions". In: *Real-Time Systems* 48.3 (2012), pp. 320–358.

[RTV10] Dennie Reniers, Alexandru Cristian Telea, and Lucian Voinea. "SolidSX: A Visual Analysis Tool for Software Maintenance". In: 2010. URL: https://api.semanticscholar.org/CorpusID:51945596.

[RV16] P. Roques and F. Vallée. "Systems Architecture Modeling with the ARCADIA Method and Capella Tool". In: (2016).

[Roq16] Pascal Roques. "MBSE with the ARCADIA Method and the Capella Tool". In: *8th European Congress on Embedded Real Time Software and Systems (ERTS 2016)*. 2016.

[Ros77] D. T. Ross. "Structured Analysis (SA): A Language for Communicating Ideas". In: *IEEE Transactions on Software Engineering* SE-3.1 (1977), pp. 16–34. DOI: 10.1109/TSE.1977.229900.

[SAE06] SAE. *AS5506B: Architecture Analysis & Design Language (AADL)*. SAE International, 2006.

[SAEa] SAE. *SAE. Architecture Analysis and Design Language V2.0 (AS5506), September 2008*. www.aadl.info.

[SAEb] SAEAS5506. *SAE AS-5506B:Architecture Analysis & Design Language (AADL)*. https://www.sae.org/standards/content/as5506b/. Accessed: 2019-02-20.

[Sai+15] Salah Eddine Saidi et al. "An ILP approach for mapping autosar runnables on multi-core architectures". In: *Proceedings of the 2015 workshop on rapid simulation and performance evaluation: methods and tools*. 2015, pp. 1–8.

[SKW00] M. Saksena, Panagiota Karvelas, and Y. Wang. "Automatic synthesis of multi-tasking implementations from real-time object-oriented models". In: *Proceedings Third IEEE International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC 2000) (Cat. No. PR00607)* (2000), pp. 360–367.

[Sei03] E. Seidewitz. "What models mean". In: *IEEE software* 20.5 (2003), pp. 26–32.

[SRL90] Lui Sha, Ragunathan Rajkumar, and John P Lehoczky. "Priority inheritance protocols: An approach to real-time synchronization". In: *IEEE Transactions on computers* 39.9 (1990), pp. 1175–1185.

[Sin+04] Frank Singhoff et al. "Cheddar: a flexible real time scheduling framework". In: *Proceedings of the 2004 annual ACM SIGAda international conference on Ada: The engineering of correct and reliable software for real-time & distributed systems using Ada and related technologies*. 2004, pp. 1–8.

[Som11] I. Sommerville. *Software Engineering*. Pearson, 2011.

[Spe06] OMG Available Specification. *Object constraint language*. 2006. URL: https://www.omg.org/spec/OCL/2.4/.

[Spe07] OMG Available Specification. "Omg unified modeling language (omg uml), superstructure, v2. 1.2". In: *Object Management Group* 70 (2007). URL: https://www.omg.org/spec/UML/2.5.1/.

[Sta88] J.A. Stankovic. "Misconceptions about real-time computing: a serious problem for next-generation systems". In: *Computer* 21.10 (1988), pp. 10–19. DOI: 10.1109/2.7053.

[Sti+11a] M. Stigge et al. "On the Tractability of Digraph-Based Task Models". In: *Euromicro Conference on Real-Time Systems*. 2011, pp. 162–171.

[Sti+11b] M. Stigge et al. "The Digraph Real-Time Task Model". In: *IEEE Real-Time and Embedded Technology and Applications Symposium*. 2011, pp. 71–80.

[TGY20] Yue Tang, Nan Guan, and Wang Yi. "Real-Time Task Models". In: *Handbook of Real-Time Computing*. Ed. by Yu-Chu Tian and David Charles Levy. Singapore: Springer Singapore, 2020, pp. 1–19. ISBN: 978-981-4585-87-3. DOI: 10.1007/978-981-4585-87-3_29-1. URL: https://doi.org/10.1007/978-981-4585-87-3_29-1.

[Tea] Cppcheck Team. *Cppcheck*. http://cppcheck.sourceforge.net/.

[TV08] Alexandru Telea and Lucian Voinea. "Solidfx: An integrated reverse engineering environment for c++". In: *2008 12th European Conference on Software Maintenance and Reengineering*. IEEE. 2008, pp. 320–322.

[Tha23] Thales. *ARCADIA: a model-based engineering method.* Accessed: 2022-08-01. 2023.

[The10] The ATESST Consortium. *EAST-ADL 2.0 Specification.* 2010.

[TC94] Ken Tindell and John Clark. "Holistic schedulability analysis for distributed hard real-time systems". In: *Microprocessing and microprogramming* 40.2-3 (1994), pp. 117–134.

[WL07] Richard Wettel and Michele Lanza. "Visualizing software systems as cities". In: *2007 4th IEEE International Workshop on Visualizing Software for Understanding and Analysis*. IEEE. 2007, pp. 92–99.

[Wil+08] Reinhard Wilhelm et al. "The worst-case execution-time problem—overview of methods and survey of tools". In: *ACM Transactions on Embedded Computing Systems (TECS)* 7.3 (2008), pp. 1–53.

[Wol23] Inc. Wolfram Research. *Wolfram*. https://www.wolfram.com/?source=nav. Accessed: 2023-10-19. 2023.

Résumé

Cette thèse porte sur la conception des systèmes embarqués temps réel critiques. Elle se concentre plus particulièrement sur la phase cruciale de déploiement, où les fonctions du système sont allouées aux tâches logicielles.

La thèse débute par une présentation approfondie du contexte des systèmes embarqués temps réel, de leurs spécificités et de leur cycle de vie logiciel. Puis, le domaine de l'ordonnancement temps réel est introduit, en détaillant les modèles de tâches, les algorithmes d'ordonnancement et les tests de validation temporelle.

Sur ces bases, deux contributions originales sont proposées pour faciliter le déploiement dans un contexte où les informations sur le comportement temporel des fonctions restent limitées. La première est la méthode RYM, une technique de pré-déploiement des systèmes mono-processeur basée sur les rythmes d'activation. Un outil logiciel est également développé pour valider cette approche. La seconde contribution est un framework pour extraire, modéliser et analyser l'architecture interne des autopilotes open-source de drones à partir du code source. Ce framework est appliqué pour adapter une technique avancée d'ordonnancement à l'autopilote Paparazzi.

Ces deux contributions apportent des réponses au problème difficile du déploiement des systèmes embarqués critiques avec des informations partielles. Elles s'appuient toutes les deux sur des développements de prototypes ouverts basés ingénierie dirigée par les modèles, permettant une adaptation rapide à d'autres outils d'analyse.

Mots-clefs : systèmes embarqués temps réel, conception, déploiement, ingénierie dirigée par les modèles, ordonnancement temps réel.

Abstract

This thesis focuses on the design of real-time critical embedded systems. It concentrates more specifically on the crucial deployment phase, where system functions are allocated to software tasks.

The thesis begins with an in-depth presentation of the context of real-time embedded systems, their specificities and their software lifecycle. Then, the field of real-time scheduling is introduced, detailing task models, scheduling algorithms and timing validation tests.

On these foundations, two original contributions are proposed to facilitate deployment in a context where information on the temporal behavior of functions remains limited. The first is the RYM method, a single-processor pre-deployment technique based on activation rhythms. A software tool is also developed to validate this approach. The second contribution is a framework to extract, model and analyze the internal architecture of open-source drone autopilots from the source code. This framework is applied to adapt an advanced scheduling technique to the Paparazzi autopilot.

These two contributions provide answers to the difficult problem of deploying critical embedded systems with partial information. They are both based on open prototyping developments based on model-driven engineering, allowing rapid adaptation to other analysis tools.

Keywords: real-time embedded systems, design, deployment, model-driven engineering, real-time scheduling.

Secteur de recherche : Informatique et applications

LABORATOIRE D'INFORMATIQUE ET D'AUTOMATIQUE POUR LES SYSTÈMES
École Nationale Supérieure de Mécanique et d'Aérotechnique
Téléport 2 – 1 avenue Clément Ader BP 40109 86961 Futuroscope Chasseneuil Cedex
Tél. +33 (0)5 49 49 80 80 Fax : +33 (0)5 49 49 80 00