Université de Bretagne Occidentale
UFR Sciences et Techniques

Étude et qualification amont d'un contrôleur domotique contraint par le temps à l'aide du langage d'architecture AADL

Mathieu Soula

Mémoire de 2e année de Master Recherche
Spécialité Logiciels pour les systèmes embarqués
Année universitaire 2013-2014

Tuteur : Frank Singhoff
Encadrant : Loïc Plassart

25 août 2014

"The computing scientist's main challenge is not to get confused by the complexities of his own making."

Edsger Wybe Dijkstra
Mathématicien et informaticien Néerlandais, prix Turing 1972

"Software and cathedrals are much the same – first we build them, then we pray. "

Samuel T. Redwine
Professeur à l'université James Madison, Harrisonburg

"Weinberg's Second Law : If builders built buildings the way programmers wrote programs, then the first woodpecker that came along would destroy civilization."

Gerald Marvin Weinberg
Informaticien et professeur de psychologie et d'anthropologie du développement logiciel
Auteur de *The Psychology of Computer Programming and Introduction to General Systems Thinking*

Remerciements

Je tiens à remercier toutes les personnes qui ont rendu ce stage possible par leur aide et leurs contributions.

Je souhaite tout d'abord remercier mon responsable de formation Jalil Boukhobza pour avoir accepté mon inscription en deuxième année de master Logiciels pour les Systèmes Embarqués.

Je remercie également Frank Singhoff, professeur à l'Université de Bretagne Occidentale, pour son soutien et pour l'aide qu'il a apportée tout au long de ce stage. Ses conseils ont été utiles afin de mener à bien ce projet.

Enfin, je remercie Loïc Plassart, président de l'association Tréflévenet, pour avoir suivi mes travaux durant ce stage. Son expérience et ses conseils avisés ont rendu possible la réalisation du travail présenté dans ce rapport.

Table des matières

1 Introduction

Un système embarqué est un système électronique et informatique autonome dédié à une tâche particulière et contenu dans un système englobant. Dans ce type d'environnement, le logiciel est fortement couplé au matériel, c'est pourquoi on parle de *logiciel embarqué*. Le domaine d'action d'un tel type de logiciel est toujours limité aux fonctions pour lesquelles il a été créé.

On assiste depuis quelques années à une explosion du nombre de systèmes embarqués destinés à un public non professionnel. Ceci est notamment visible dans le secteur de la téléphonie et de la domotique où les constructeurs cherchent constamment à créer de nouveaux équipements proposant des fonctionnalités novatrices. Les systèmes devenant de plus en plus complexes, leur coût de production tend à augmenter sensiblement. De nouvelles méthodes de conception ont donc été pensées afin de réduire ces coûts. Les frameworks de description d'architecture proposent par exemple une notation formelle permettant de décrire un système logiciel en tenant à la fois compte de l'aspect logiciel et de l'aspect matériel. Grâce à des outils d'analyses et de vérifications, ces solutions permettent de déplacer une partie du travail de qualification en amont des phases de conception.

Loïc Plassart de l'association TréflévéNet [Tréflévénet, 2014], travaille à la conception d'un contrôleur domotique. Ce contrôleur devra assurer plusieurs fonctions (affichage d'une interface graphique, commande de chauffage, commande d'éclairage, pilotage d'entrées-sorties, gestion d'un bus de terrain, etc.). Il souhaite désormais pouvoir qualifier son système pour s'assurer du bon respect des exigences fonctionnelles et non fonctionnelles. Cette qualification a la particularité d'être réalisée avant l'étape de conception du contrôleur.

L'objectif de mon stage consiste à réaliser une étude qualificative du contrôleur domotique. Concrètement, mon travail consistera tout d'abord à proposer un modèle d'architecture du contrôleur. Pour cela, le langage de description d'architecture AADL sera utilisé. Dans un second temps, je devrai réaliser un ensemble de tests afin de vérifier si le modèle défini répond ou non aux différentes exigences temporelles du système. J'utiliserai pour cela le moteur de Cheddar v3, un outil d'analyse de performances développé à l'Université de Bretagne Occidentale et qui peut être utilisé sur des modèles d'architecture écrits avec le langage AADL.

Le mémoire est structuré de la manière suivante. Le chapitre 2 présente un état de l'art sur les langages de description d'architecture logicielle, introduit le langage AADL, et rappelle quelques bases sur la théorie d'ordonnancement temps-réel. Le chapitre 3 décrit le travail de modélisation du contrôleur domotique. Puis, le chapitre 4 présente les tests d'ordonnançabilité réalisés sur ce modèle. Enfin, le mémoire se conclut au chapitre 5 dans lequel sont présentées les perpectives d'évolutions possibles de mon travail.

2 Contexte de la description d'architecture logicielle

Ce chapitre a pour but de délimiter les différents domaines concernés par ce stage. Il s'agit ici de constituer un état de l'art de la description d'architecture, plus particulièrement dans le domaine des logiciels. Dans une première partie, nous présentons un ensemble de définitions afin de mieux appréhender le domaine des langages de description d'architecture. Puis dans une seconde partie, nous proposons une introduction au langage de description d'architecture AADL. Puis dans une troisième partie, nous donnons quelques notions de la théorie d'ordonnancement temps-réel qui seront utiles à la lecture de ce rapport. Enfin, nous terminons ce chapitre en discutant de quelques travaux de modélisation AADL dans une optique de qualification temporelle de systèmes.

Dans le reste de ce document, nous utiliserons l'acronyme **AD** pour faire référence au terme *description d'architecture*, et l'acronyme **ADL** pour faire référence au terme *langage de description d'architecture*.

2.1 Notions des langages de description d'architecture

Cette section présente au lecteur le domaine des langages de description d'architecture. Après avoir présenté le standard ISO/IEC/IEEE 42010 qui propose un ensemble de définitions liées au domaine, nous discutons des principales caractéristiques d'un ADL.

2.1.1 Définition de la description d'architecture

L'ISO/IEC/IEEE 42010 [42010, 2011] est un standard international intitulé *Systems and software engineering - Architecture description*[1]. Publié en 2011, il est le résultat d'une révision du standard IEEE Std 1471-2000 [Group, 2000] *IEEE Recommended Pratice for Architectural Description of Software-Intensive Systems*[2] réalisée conjointement par ISO et IEEE. Ce standard propose un ensemble de modèles conceptuels de définitions et de concepts ayant trait à l'AD. Ces modèles se présentent sous la forme de diagrammes de classes UML détaillant des classes d'entités et leurs relations. La figure 2.1 [42010, 2011]

1. Ingénierie des systèmes et des logiciels – Description de l'architecture
2. Pratiques recommandées par le IEEE pour la description architecturale des systèmes exigeant beaucoup de logiciels

présente le contexte de la description d'architecture. Les différentes entités sont détaillées ci-dessous.

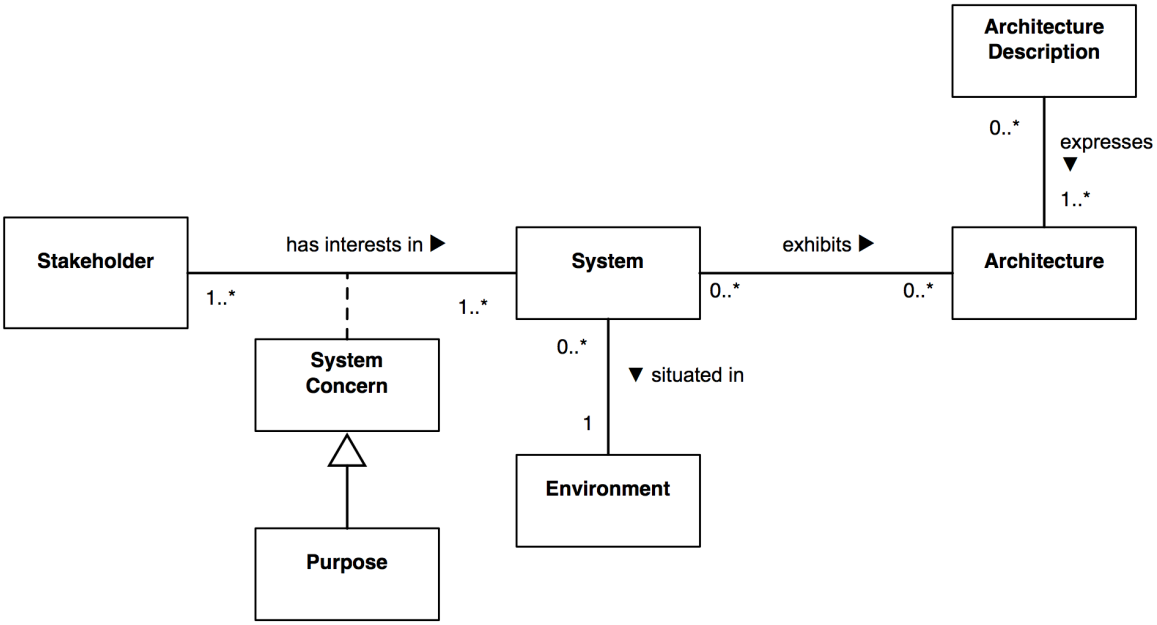

FIG. 2.1 – Termes et concepts de la description d'architecture

System – Le standard ne définit pas précisément à quoi un *système* se rapporte. Il s'agit d'une abstraction permettant d'adapter l'AD au plus grand nombre de domaines. L'utilisateur de l'AD se charge de définir à quoi correspond le système dont il décrit l'architecture (ex. : une entreprise, un service, ou encore une application).

Environment – Tout système doit être considéré au sein d'un *environnement* qui l'influence. Ces influences peuvent être de tout ordre (politique, budgétaire, techniques, etc.) et ont un impact direct sur l'architecture à décrire. Ces influences ont pour nom *concerns*[3] dans le standard. Des centres d'intérêt possibles peuvent être : le coût, le besoin d'interopérabilité, la sécurité, etc.

Architecture – La définition donnée par le standard est *concept fondamental ou propriétés d'un système, représenté au sein de son environnement, et incarné par ses élements, leurs relations, et par les fondements de leur conception et leur évolution.* Un même système peut être représenté par une ou plusieurs architectures, et une architecture peut décrire un ou plusieurs systèmes selon le niveau d'abstraction.

Architecture description – Une *description d'architecture* est le produit d'un travail dont le but est d'exprimer l'architecture d'un système. Elle sert de base de référence aux parties prenantes et aux architectes pour comprendre, analyser et comparer des architectures. Une AD peut se présenter sous différentes formes : documents, ensemble de modèles, etc.

Stakeholder – Les *parties prenantes* caractérisent toute personne, groupe de personnes, ou organisation ayant des *centres intérêt* sur le système à décrire : client, propriétaire, architecte, développeur, etc.

3. Centres d'intérêt

2.1.2 Définition de langage de description d'architecture

Le standard ISO/IEC/IEEE 42010 définit un langage de description d'architecture comme toute forme d'expression utilisée dans le contexte de la description d'architecture. Dans cette partie, nous détaillons les principaux objectifs d'un ADL dans le cadre de l'architecture logicielle, puis nous décrivons les principales caractéristiques de ce type de langage.

Pour Allen, un ADL a principalement 3 objectifs : 1) proposer une notation formelle claire et précise, 2) permettre l'analyse des systèmes décrits (si possible grâce à des outils automatisés), et 3) pouvoir s'adapter au vocabulaire de l'architecte [Allen, 1997].

Avant l'arrivée des ADLs, les systèmes étaient généralement décrits à l'aide de "diagrammes de boîtes et de flèches". Dans cette configuration, les concepts sont modélisés graphiquement et servent de référence aux parties prenants pour communiquer. Le problème avec ce mode de représentation non formelle est qu'il est soumis à l'interprétation des participants. Par conséquent, des erreurs de compréhension peuvent subvenir. Une méthode de notation claire et compréhensible par tous est rapidement devenue une nécessité.

L'utilisation d'une notation formelle induit la possibilité d'appliquer des outils automatisés pour réaliser diverses analyses et opérations sur les modèles exprimés. Par exemple, des *parsers*[4] peuvent être utilisés afin de détecter toute erreur de syntaxe sur les modèles, et des compilateurs peuvent servir à repérer des erreurs de sémantique. Des outils plus perfectionnés peuvent être utilisés pour des analyses plus poussées comme des tests d'ordonnançabilité (nous le verrons plus loin), ou encore pour générer du code exécutable à partir de modèles.

Enfin, comme nous l'avons précédemment expliqué, la description d'architecture ne se cantonne pas à un seul domaine. Il en est de même pour les ADLs. Ainsi, plus un ADL permettra à un architecte d'exprimer une architecture sans effort de traduction, plus il aura de chance d'être adopté et donc réutilisé.

La figure 2.2 [42010, 2011] présente le contexte d'un langage de description d'architecture. Les différentes entités sont détaillées ci-dessous.

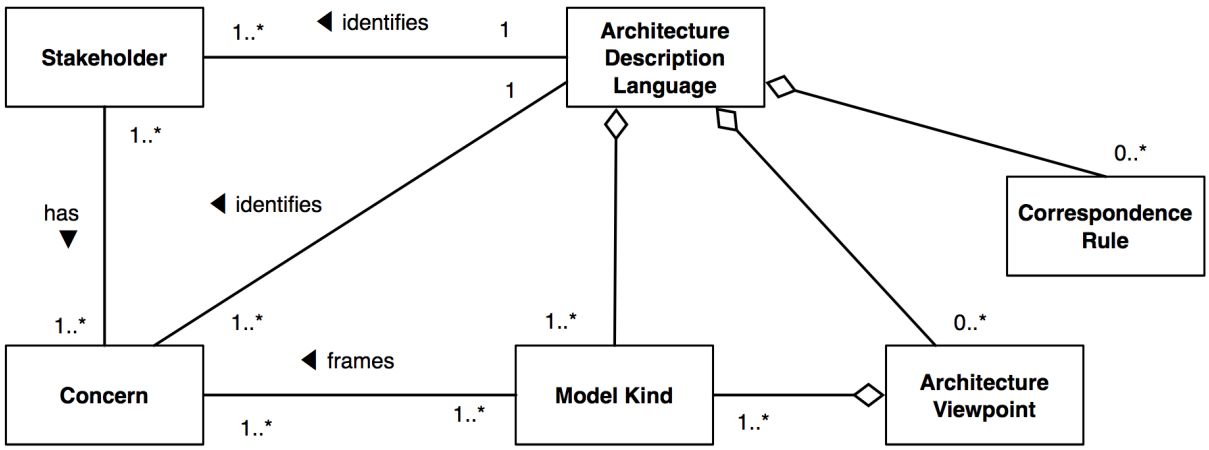

Fig. 2.2 – Termes et concepts des langages de description d'architecture

4. Outils d'analyses syntaxiques

Architecture viewpoint – Un *point de vue d'architecture* est un ensemble de conventions pour construire, interpréter, analyser et utiliser une *vue d'architecture*. Une vue d'architecture exprime l'architecture d'un système selon le point de vue d'une ou de plusieurs parties prenantes. Le point de vue d'architecture décrit entre autres : des types de modèles, des langages et des notations, des méthodes de modélisation et des techniques d'analyses permettant de cibler un ensemble d'intérêts précis.

Model kind – Un *type de modèle* définit les conventions d'un type de *modèle d'architecture*. Les vues d'architecture sont constituées d'un ou plusieurs modèles d'architecture. Ceux-ci fournissent des méthodes pour partager des détails entre vues d'architecture, et permettent aussi l'utilisation de plusieurs types de notations à l'intérieur d'une même vue.

Correspondence rule – Dans une description d'architecture, chaque entité est représentée par un élément. Chaque élément peut être en relation avec d'autres éléments, on parle alors de *correspondance*. Les correspondances sont notamment utilisées pour exprimer les relations d'intérêts à l'intérieur d'une ou de plusieurs descriptions d'architecture. Les *règles de correspondance* sont alors utilisées pour renforcer ces relations.

2.1.3 Composition d'un langage de description d'architecture

Les blocs constitutifs d'un ADL sont les *composants*, les *connecteurs*, et les *descriptions de configuration* (ou *déploiements*). Tout ADL doit fournir des moyens pour pouvoir les spécifier explicitement. De plus, on attend d'un ADL qu'il soit fourni avec un ensemble d'*outils* permettant de réaliser des analyses plus ou moins complexes sur les modèles décrits. Ces outils peuvent par exemple proposer des fonctionnalités de spécification de modèles, de vues multiples, de génération de code, etc.

Les premiers langages de description d'architecture sont apparus au début des années 90 et se sont rapidement multipliés. Des tentatives de classification et de comparaison de ces langages ont alors été proposées afin de combler le manque de définition claire et précise. Parmi ces publications, nous pouvons citer celle de Nenad Medvidovic intitulée *A Classification and Comparison Framework for Software Architecture Description Languages*[5] [Medvidovic, 1997]. Ce rapport technique propose une taxonomie détaillée afin de classer les fonctionnalités des ADLs. Quoique ancien, ce rapport est encore largement cité dans les publications associées au domaine de la description d'architecture.

Les composants – Un composant modélise une unité de calcul ou de stockage de données (ex. : une fonction procédurale, une application). Ils constituent les éléments centraux d'une architecture logicielle. Ils peuvent être simples ou bien composés d'un ensemble de sous-composants, on parle alors de *composite*. Un composant a les caractéristiques suivantes :

- une **interface** : l'ensemble des points d'interaction de ce composant avec le reste du monde ; spécifie les services offerts par le composant, ainsi que les services requis de la part des autres composants de l'architecture. Un point d'interface est généralement appelé un *port*.
- un **type** : le modèle abstrait à partir duquel il est possible d'instancier différentes implantations d'un même type de composant (améliore la réutilisabilité du composant dans les modèles).

5. Un cadre pour la classification et la comparaison des langages de description d'architecture logicielle

- une **sémantique** : dont le rôle consiste à compléter l'interface en enrichissant le modèle avec l'utilisation de propriétés afin de permettre des analyses plus fines, renforcer les contraintes, et assurer la cohérence du modèle sur différents niveaux d'abstraction.
- des **contraintes** : des propriétés et/ou assertions dont la violation rend le système décrit incompatible avec les attentes de ses parties prenantes (renforcement des conditions d'utilisation, établissement des dépendances entre sous-composants).
- une **évolutivité** : on attend d'un composant de pouvoir être réutilisé dans d'autres modèles (par des techniques comme le sous-typage ou le raffinement).
- des **propriétés non fonctionnelles** : des propriétés qui permettent notamment de simuler le comportement attendu du composant à l'exécution, de renforcer les contraintes, de préciser les critères de performances, etc.

Les connecteurs – Un connecteur modélise les interactions entre composants et les règles qui gouvernent ces interactions. Contrairement aux composants, les connecteurs ne sont pas traduits en unités de calcul dans le système implanté. Leur implantation peut prendre différentes formes : périphériques de routage de messages, variables partagées, mémoires tampons, structures de données dynamiques, etc. Un connecteur a les caractéristiques suivantes :

- une **interface** : l'ensemble des points d'interaction entre le connecteur et les composants auxquels il est lié. Ces points d'interface sont généralement appelés *pôles*. On distingue les *pôles d'entrée* des *pôles de sortie*.
- un **type** : le modèle abstrait à partir duquel il est possible d'instancier différentes implantations du même type de connecteur (améliore la réutilisabilité du connecteur dans les modèles). Les types de connecteur peuvent représenter les protocoles de communication.
- une **sémantique** : dont le rôle consiste à compléter l'interface en enrichissant le modèle avec l'utilisation de propriétés afin de permettre des analyses plus fines, renforcer les contraintes, et assurer la cohérence du modèle entre les différents niveaux d'abstraction.
- des **contraintes** : des propriétés et/ou assertions dont la violation rend le système décrit incompatible avec les attentes de ses parties prenantes (renforcement des conditions d'utilisation, établissement des dépendances entre sous-composants).
- une **évolutivité** : on attend d'un connecteur de pouvoir évoluer avec les mises à jour du protocole qu'il modélise (par des techniques comme le sous-typage, ou le raffinement).
- des **propriétés non fonctionnelles** : des propriétés qui permettent notamment de simuler le comportement attendu du connecteur à l'exécution, de renforcer les contraintes, de préciser des critères de performances, etc. Les propriétés non fonctionnelles d'un connecteur peuvent avoir une influence sur le choix du type de connecteur (ex. : bus de message) à implanter sur les processeurs.

Les descriptions de configuration – Une description de configuration correspond à un graphe connecté de composants et de connecteurs qui décrit la structure de l'architecture d'un système. Elle permet de déterminer si les composants appropriés sont bien connectés, si leurs interfaces correspondent, si les connecteurs supportent le bon protocole de communication, et si les sémantiques combinées fournissent le comportement attendu.

Une description de configuration a les caractéristiques suivantes :

- des **spécifications compréhensibles** : la compréhension des configurations facilite la communication entre les parties prenantes. Un syntaxe claire et précise couplée à un langage graphique interchangeable correspond à une solution pour répondre à cet objectif.
- une **hiérarchie de composants** : cette hiérarchisation permet de structurer l'architecture. La décomposition de l'architecture permet d'obtenir différents niveaux d'abstraction et donc une meilleure expressivité du modèle.
- une **hétérogénéité des éléments** : l'objectif des architectures logicielles est de faciliter le développement de systèmes de grande taille dans lesquels coexistent une grande variété de composants à la granularité variable, ainsi que des connecteurs décrivant différents protocoles de communication. La capacité d'un langage à représenter cette variété d'éléments améliore l'expressivité du modèle et permet de s'adapter à tous types de domaine.
- des **contraintes** : qui permettent de décrire les dépendances nécessaires entre composants et connecteurs au sein d'une configuration. Les contraintes de description de configuration permettent d'établir les contraintes locales du système.
- la capacité de **raffiner le modèle** et de **tracer ces modifications** : l'objectif d'un ADL est de décrire une architecture qui sera ensuite traduite en code exécutable. Ceci est rendu possible par le raffinement progressif des composants jusqu'au niveau le plus élémentaire. Le traçage de ces raffinements permet la navigation entre les différents niveaux d'abstraction.
- la capacité de **changer d'échelle**, d'**évoluer**, et de **dynamiser** les modèles : les modèles ne sont pas statiques et évoluent en fonction des besoins (nouvelles spécifications, changement de protocole de communication, etc.). La possibilité de s'adapter à ces changements améliore la réutilisabilité des différents éléments.
- des **propriétés non fonctionnelles** : des propriétés qui permettent notamment de simuler le comportement attendu du système décrit, de renforcer les contraintes, de mapper les blocs d'architecture sur les processeurs, et d'aider à la gestion de projet.

Les outils de support – Les outils de support d'un ADL constituent sa valeur ajoutée. Ces outils sont destinés à réaliser différents types d'analyse, de vérification, de simulation, ou d'évaluation sur les performances. Plus que les possibilités offertes par le langage, c'est la présence et surtout la variété des fonctionnalités offertes par ses outils qui conditionnera l'adoption d'un ADL par les utilisateurs [Malavolta et al., 2013]. Les outils d'un ADL peuvent être classés par :

- la capacité de **spécifier des architectures** : plus qu'un simple éditeur de texte ou un outil de modélisation graphique, les outils doivent apporter des fonctionnalités d'aide à la notation (on peut distinguer les outils *pro-actifs* qui s'adaptent en fonction de l'état courant du modèle des outils *réactifs* qui détectent les erreurs au cours de la modélisation).
- la capacité de proposer **différentes vues** du même modèle : cela afin de permettre aux différentes parties prenantes d'intervenir sur le point qui les intéresse. Généralement, les outils proposent deux types de vue : textuelle et graphique. D'autres types de vue sont possibles : visualisation de la hiérarchie des composants, visualisation du processus de développement, visualisation animée du déroulement d'une simulation, etc.
- les différents types d'**analyse** disponibles : ceux-ci doivent couvrir un large éventail

allant de la correction des erreurs de syntaxe (via des parsers) et de sémantique (via des compilateurs), à la prévention de deadlocks potentiels, ou bien la réalisation de tests de faisabilité.
- la cohérence du modèle à travers ses différents niveaux de **raffinement**.
- la **génération de code** : l'un des principaux objectifs des ADLs est de permettre la génération automatique de code à partir de modèle d'architecture. De cette manière, il est possible d'éviter les erreurs que le développeur pourrait introduire en implantant "à la main" un modèle. Cette fonctionnalité est retrouvée dans la plupart des frameworks d'ADLs (Aesop, C2, Darwin, Rapid, etc.).
- la capacité de gérer **dynamisme** des modèles.

2.2 Présentation de AADL

La Society of Automotive Engineers (SAE) est une association internationale regroupant différents types d'acteurs du milieu automobile (étudiants, ingénieurs, chefs d'entreprise, etc.) qui échangent autour des idées relatives à l'automobile. En novembre 2004, elle publie le standard aérospatial AS5506 intitulé *Architecture Analysis and Design Language*[6], ou AADL. Il s'agit d'un langage de description d'architecture basé sur MetaH [Feiler et al., 2006]. AADL a été pensé pour permettre la représentation d'architectures de systèmes embarqués.

Une description d'architecture AADL différencie la partie logicielle de la partie matérielle d'un système. Côté logiciel, on peut distinguer les composants de type **process**, **thread**, **thread group**, **data**, et **subprogram**. Côté matériel, on distingue les composants de type **processor**, **memory**, **bus**, et **device**. Enfin, il existe le composant de type **system** qui correspond à un composite utilisé pour constituer des blocs logiques de sous-composants, et ainsi architecturer un modèle.

Cette partie traite des différents éléments du langage AADL, puis présente quelques outils se basant sur ce langage.

2.2.1 Eléments du langage

Dans cette partie, nous détaillons les composants permettant de décrire la partie logicielle d'un système, puis ceux utilisés pour décrire la partie matérielle. Enfin, nous présentons le composant system.

process – Le composant process décrit un espace d'adressage protégé, un espace de partitionnement où tout ce qui est contenu est protégé des autres composants du système. Cet espace d'adressage contient :
- les images binaires exécutables (code exécutable et données) associées au process ;
- les images binaires exécutables associées aux sous-composants du process ;
- les sous-programmes offrant des services (code exécutable) et des données référencés par les composants externes.

Le composant process accepte 3 types de sous-composants : thread, thread group, et data. Contrairement à un processus POSIX, un process ne contient pas de thread de manière

6. Langage de conception et d'analyse d'architecture

implicite. Pour représenter un composant actif, il est nécessaire de déclarer un thread, ou un thread group, comme sous-composant du process.

thread – Le composant thread décrit un composant actif (comme un thread POSIX). Les threads AADL peuvent avoir différents modes opératoires. Chaque mode peut décrire un comportement et des valeurs de propriétés différentes. Le composant thread n'accepte qu'un seul type de sous-composant : data. Il est possible de déclarer un appel à un subprogram [7].

thread group – Le composant thread group décrit un composant abstrait regroupant un ensemble de threads, de data, et de thread groups au sein d'un process. Les thread groups offrent une séparation des points d'intérêts en définissant une référence unique à un ensemble de threads et de data. Le composant thread group accepte 3 types de sous-composants : thread, thread group, et data

data – Le composant data représente des données statiques (ex : des données numériques, un texte source) et des types de données d'un système. Les déclarations de composant data sont notamment utilisées pour représenter :

- les types de données d'une application (ex. : types de données échangées entre les ports, paramètres) ;
- les sous-structures de types de données (par composition) ;
- les instances de données.

Le composant data n'accepte qu'un seul type de sous-composant : data. Il est donc possible de déclarer des structures hiérarchiques de données.

subprogram – Le composant subprogram représente un code source exécutable de manière séquentielle, c'est-à-dire un composant pouvant être invoqué avec ou sans paramètre, et qui travaille sur des données ou fournit des fonctionnalités aux composants qui l'utilisent. La signature du subprogram et de ses paramètres est déclarée comme sous-composant. Seuls le thread et le subprogram peuvent déclarer un appel à un subprogram. Le composant subprogram ne peut pas contenir de sous-composant.

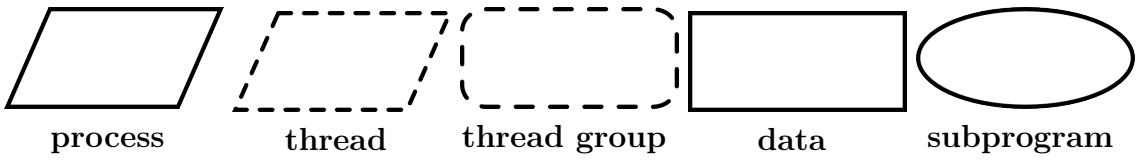

FIG. 2.3 – Représentation graphique des composants logiciels AADL

processor – Le composant processor est une abstraction du matériel et de la partie applicative associée responsable de l'ordonnancement et de l'exécution d'un ensemble de threads. Il peut embarquer une partie logicielle (ex. : un système d'exploitation) chargée de l'ordonnancement ainsi que d'autres fonctionnalités destinées au support des threads. Le composant processor n'accepte qu'un seul type de sous-composant : memory.

7. Sous-programme

memory – Le composant memory représente un composant de stockage de données ou de code exécutable. Il peut représenter une mémoire à l'intérieur d'un processor ou d'une unité de plateforme d'exécution séparée. Une mémoire est connectée à un processor via un bus. Le composant memory n'accepte qu'un seul type de sous-composant : memory.

bus – Le composant bus représente le matériel et les protocoles de communication associés qui rendent possibles les interactions entre les composants d'une plateforme d'exécution (c.-à-d. : memory, processor, et device). Les bus peuvent être connectés directement à d'autres bus pour représenter des communications complexes entre réseaux. Le composant bus ne peut pas contenir de sous-composant.

device – Le composant device représente une entité qui sert d'interface entre une application système et l'environnement extérieur. Les devices peuvent avoir des comportements complexes et contenir des composants matériels qui ne sont pas représentés explicitement dans le modèle. Le composant device ne peut pas contenir de sous-composant.

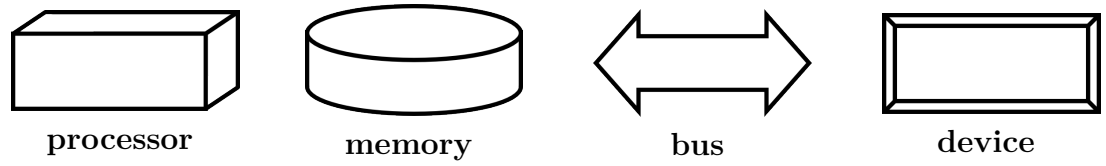

FIG. 2.4 – Représentation graphique des composants matériels AADL

system - Le composant system est un composite qui peut contenir des composants logiciels, matériels, ou systèmes. Ce type de composant est utilisé afin de faciliter la représentation hiérarchique des composants d'une architecture. Le composant system est placé tout en haut de la hiérarchie des composants ; il peut donc accepter les sous-composants suivants : data, process, processor, memory, bus, device, et system.

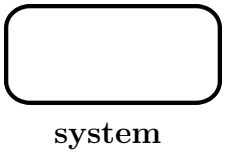

FIG. 2.5 – Représentation graphique du composant AADL system

2.2.2 Extension du langage

Il est possible d'étendre le langage de base par la déclaration de bibliothèques d'annexes. Ces extensions permettent notamment de réaliser des analyses personnalisées via l'utilisation de modèles et d'abstractions spécialisés [Bozzano et al., 2010].

2.2.3 Quelques outils

Dans cette partie, nous présentons quelques outils existants pour réaliser des modèles AADL ainsi que différents types d'analyse.

Stood – Stood est un outil de modélisation qui supporte le langage AADL. De plus, il permet d'appliquer la méthodologie HOOD [8] [Dissaux and Singhoff, 2008]. HOOD est une méthode de modélisation créée par l'Agence Spatiale Européenne en 1987. Elle permet de décomposer un problème logiciel en une hiérarchie d'objets, de classes, et d'opérations. La méthologie propose entre autres un langage textuel et un langage graphique. Avec Stood, il est possible de gérer complètement un projet logiciel en construisant des bibliothèques de composants réutilisables, de faire du rétro-engineering de code, et de modéliser une application temps-réel et sa plateforme d'exécution. Stood propose aussi une fonctionnalité de génération de code AADL à partir des modèles spécifiés.

Ocarina – Ocarina est une suite d'outils développée par Télécom Paris. Cette application est construite autour d'un noyau qui fournit un ensemble de fonctionnalités de manipulation de modèles AADL. Parmi ces fonctionnalités, on peut notamment citer l'analyse syntaxique et la représentation graphique de modèles AADL (utilisation de Dia), ainsi que des fonctionnalités de génération de code pouvant être déployé sur différentes plateformes.

Cheddar – Cheddar est un framework qui fournit des outils de simulation permettant de réaliser des tests selon différents critères de performances (contraintes temporelles, dimensionnement des ressources, etc.) [Singhoff et al., 2004]. Cheddar se base entre autres sur la théorie d'ordonnancement temps-réel pour évaluer si un système répond à un ensemble de contraintes temporelles. Il est possible de spécifier une application en déclarant un ensemble de processeurs, de tâches, de buffers, de ressources partagées, et de messages. Des tests de faisabilité peuvent alors être réalisés. Cheddar v3 propose une fonctionnalité de parsing de modèles AADL v1. Il est donc possible de charger un modèle d'application AADL et de réaliser des simulations sur l'ordonnancement de ce modèle.

AADLInspector – AADLInspector est un framework de traitement de modèles AADL développé et édité par la société Ellidiss. Son objectif est de fournir une solution légère et évolutive permettant de réaliser des analyses statiques et dynamiques d'architectures écrites en AADL, et de se connecter facilement sur des outils de vérification de code AADL ou de génération de code exécutable compatibles. AADLInspector se base sur le moteur de Cheddar v3 pour réaliser des tests de faisabilités et des simulations d'ordonnancement. Il rend aussi possible l'analyse de modèles AADL v2 grâce à un outil de formattage embarqué qui transforme du code AADL v2 en code AADL compatible Cheddar v3.

OSATE – Osate est un plugin Eclipse permettant de gérer des projets de modélisation AADL. Gratuit, il permet de rapidement créer des modèles AADL v2, et propose un ensemble d'outils d'analyse comme un parser syntaxique ou un analyseur sémantique. Il propose aussi une fonctionnalité de modélisation passant par la notation graphique du langage AADL. Parmi les autres fonctionnalités offertes par Osate, nous pouvons citer un outil d'analyse de flux de données. Les flux de données permettent de décrire une communication bout-à-bout entre composants d'un modèle [Hudar and Feiler, 2007]. Il est donc possible de réaliser des tests sur les temps de transmission d'un message à travers tout un système.

8. Hierarchic Object-Oriented Design

2.3 Théorie d'ordonnancement temps-réel

John Stankovic [9] propose la définition suivante pour décrire un système temps-réel : *Les systèmes temps-réel sont des systèmes pour lesquels l'exactitude ne se mesure pas seulement sur les résultats des calculs logiques, mais aussi sur le temps sur lequel ces résultats sont produits* [Stankovic, 1988]. La théorie de l'ordonnancement temps-réel regroupe un ensemble d'algorithmes permettant de déterminer la manière selon laquelle un ensemble de tâches se partagent du temps de calcul. Elle propose aussi un ensemble de méthodes analytiques appelées *tests de faisabilité* et *tests d'ordonnancement*. Ces tests sont utiles pour déterminer le comportement d'un système avant son implantation/exécution [Singhoff, 2014].

Après avoir présenté le modèle canonique d'une tâche, nous donnons un aperçu des outils mis à disposition des architectes système par la théorie d'ordonnancement temps-réel.

2.3.1 Modèle canonique d'une tâche

Une tâche est constituée d'une séquence d'instructions, d'un ensemble de données, et d'un contexte d'exécution. On distingue principalement 2 types de tâches : les tâches *périodiques* et *apériodiques*. Dans un système temps-réel, ce sont les tâches périodiques qui sont considérées comme critiques. Par conséquent, il est important d'en satisfaire les propriétés d'ordonnancement.

Une tâche i, qu'elle soit périodique ou apériodique, possède un ensemble de propriétés d'ordonnancement :

- une date d'arrivée dans le système S_i ;
- une date de premier déclenchement R_i ;
- une période d'activation P_i qui correspond au temps écoulé entre deux déclenchements de la tâche ;
- une deadline D_i qui correspond au délai maximum acceptable pour son exécution ;
- une capacité C_i qui correspond au pire temps CPU nécessaire à son exécution ;
- une priorité utilisée par l'ordonnanceur pour sélectionner la tâche parmi les tâches disponibles.

La figure 2.6 [Pouiller, 2011] permet de visualiser ces différentes propriétés.

La théorie d'ordonnancement temps-réel propose plusieurs méthodes pour attribuer une priorité à une tâche d'un système. Par exemple, il est possible de se baser sur la criticité ou l'urgence de la tâche. On peut aussi considérer les dépendances entre tâches et faire en sorte qu'une tâche qui a besoin du résultat d'une autre tâche possède une priorité moins élevée. Il existe aussi plusieurs algorithmes d'ordonnancement parmi lesquels nous pouvons citer **Rate-Monotonic** [Liu and Layland, 1973], **Early Deadline First** (EDF), **FIFO**, **LIFO**, et **Round-Robin**. Rate-Monotonic est généralement utilisé dans les problématiques d'ordonnancement à priorité fixe.

9. Professeur à l'Université du Massachussets de Amherst qui a beaucoup travaillé sur les systèmes temps-réel

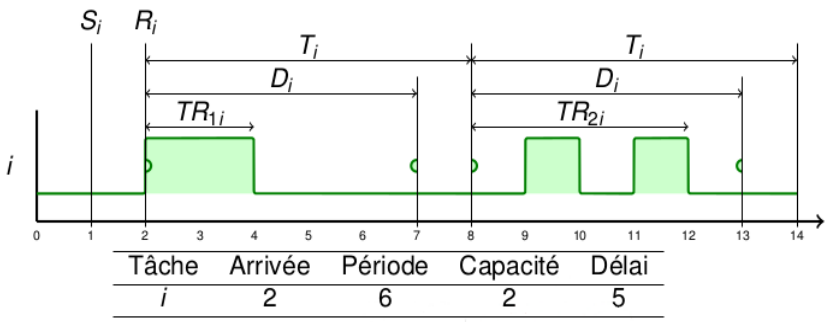

FIG. 2.6 – Propriétés d'ordonnancement d'une tâche

2.3.2 Méthodes de vérification de l'ordonnançabilité d'un système

La théorie d'ordonnancement propose un ensemble de méthodes afin de tester l'ordonnançabilité d'un système. Dans le cadre de l'ordonnancement à priorité fixe dans un système préemptif, le calcul du taux d'occupation du processeur est suffisant pour vérifier si un jeu de tâches donné est ordonnançable. Le calcul du pire temps d'exécution ou encore la simulation d'ordonnancement sur la période fondamentale (aussi appelée *hyperpériode*) peuvent aussi être utilisés pour vérifier l'ordonnançabilité d'un système.

La méthode du calcul du taux d'occupation du processeur [Liu and Layland, 1973] est une méthode utilisée pour vérifier l'ordonnançabilité d'une tâche. Elle repose sur l'équation

$$U = \sum_{i=1}^{n} \frac{C_i}{P_i} \leq n\left(2^{\frac{1}{n}} - 1\right) \tag{2.1}$$

où n est le nombre de tâches du système, C_i est la capacité de la tâche i, et P_i est période de la tâche i. La validation de cette équation est une condition suffisante pour confirmer l'ordonnançabilité d'un système.

Le calcul du pire temps d'exécution d'une tâche [Audsley et al., 1993] permet de déterminer le délai entre l'activation d'une tâche et sa terminaison. Un jeu de tâches est ordonnançable si toutes les échéances sont respectées. Cette méthode repose sur l'équation

$$r_i = C_i + \sum_{j \in hp(i)} \lceil \frac{r_i}{P_j} \rceil C_j \tag{2.2}$$

où r_i est le pire temps de réponse de la tâche i, C_i est la capacité de la tâche i, $hp\,(i)$ est l'ensemble des tâches qui ont une plus grande priorité que la tâche i, C_j est la capacité de la tâche j, et P_j est la période de la tâche j. L'évaluation des temps d'exécution d'un jeu de tâches est une condition suffisante et nécessaire pour vérifier l'ordonnançabilité d'un système.

Simulation d'ordonnancement sur l'hyperpériode

L'hyperpériode (ou période fondamentale) correspond au plus petit multiple commun de toutes les périodes d'un jeu de tâches. La simulation de l'ordonnancement d'un jeu de tâches sur l'hyperpériode permet d'avoir les résultats exacts du comportement du système et constitue une condition nécessaire et suffisante pour en vérifier l'ordonnançabilité.

2.4 Description d'architecture et qualification temps-réel

Dans cette partie, nous présentons trois travaux proposant des solutions différentes pour qualifier temporellement un système décrit en langage AADL. Le premier travail décrit une méthode de traduction de modèles AADL en modèles BIP pour utiliser les fonctionnalités offertes par le framework BIP. Le deuxième travail s'intéresse à la notion de flux afin d'évaluer le temps de communication bout-à-bout d'un message dans un système composé de plusieurs composants. Enfin, le dernier travail présente une méthode qui se base sur l'utilisation de Cheddar pour réaliser des tests de faisabilité sur un système décrit à partir du langage AADL.

Le sujet de thèse de Mohamed Y. Chkouri a porté sur la définition d'une méthodologie de traduction de modèles AADL en langage BIP [10] [Chkouri, 2010]. BIP est un framework utilisé pour décrire des architectures rigoureuses. En plus d'un langage spécifique, il fournit un ensemble d'outils permettant de construire des systèmes complexes en associant le comportement d'un ensemble de composants atomiques. Ce comportement est décrit comme un réseau de Petri étendu avec des données et des fonctions écrites en langage C. BIP dispose notamment d'un moteur temps-réel qui exécute des programmes BIP sur une plateforme cible. La transformation du modèle AADL en description BIP est réalisée grâce à une extension de la syntaxe du langage AADL spécialement créée. Chkouri valide sa méthodologie à travers un exemple de traduction d'un modèle AADL d'ordinateur de vol.

Dans leur guide pratique sur la création de modèles AADL pour les systèmes de contrôle, John Hudak et Peter Feiler proposent d'utiliser les flux AADL pour évaluer le temps de communication bout-à-bout d'une information [Hudar and Feiler, 2007]. Pour décrire cette méthode, ils proposent de considérer un modèle de système de contrôle d'un véhicule. Les flux sont des outils du langage AADL permettant d'introduire les notions de *flow source* [11], *flow path* [12], et de *flow sink* [13]. Ainsi, il est possible de considérer la transmission d'une information à travers un ensemble de composants, et de réaliser des mesures sur les temps de transmission. Ce guide permet aussi de découvrir certaines fonctionnalités offertes par le framework Osate en ce qui concerne l'étude des flux d'un système. L'avantage de cette méthode est qu'il est possible de regrouper l'ensemble des opérations de modélisation et d'évaluation au niveau d'un seul outil : Osate.

Dans leur publication *Développement de systèmes à l'aide d'AADL – Ocarina/Cheddar*, Jérome Hugues et Frank Singhoff proposent une méthode de caractérisation temps-réel de modèles utilisant les outils Ocarina et Cheddar [Hugues and Singhoff, 2009]. Cette approche consiste à construire le modèle AADL d'un système, et à réaliser des analyses d'ordonnancement à partir de Cheddar. Cheddar permet de compléter la description des propriétés d'ordonnancement des entités du modèle grâce au **property set** `Cheddar_Properties`. Cheddar intègre un analyseur de modèles AADL qui se base sur Ocarina. Il fournit ses résultats d'analyse de l'ordonnancement dans une fenêtre comportant deux parties : un chronogramme décrivant l'ordonnancement des threads pendant l'exécution du système, et les différents critères de performances (temps de réponse des threads, délais maximaux d'attente pour l'accès aux ressources partagées, etc.). L'exemple

10. Behavior Interaction Priority
11. Source du flux
12. Parcours du flux
13. Réceptacle du flux

donné se base sur la définition d'un modèle de radar constitué de plusieurs composants : un processeur et un processus hébergeant plusieurs threads.

Dans le cadre de cet état de l'art, nous n'avons pas trouvé de publication traitant de la description d'architecture de système domotique dans une optique de caractérisation des contraintes temporelles.

2.5 Conclusion

Nous avons pu voir que la description d'architecture regroupe un grand nombre de concepts et permet de s'adapter à tout type de domaine. AADL, un langage de description d'architecture, permet de décrire des modèles de systèmes temps-réel grâce à la distinction des parties matérielle et logicielle. Les outils développés autour de ce langage autorisent toutes sortes d'analyses, notamment des analyses de l'ordonnancement. Dans le cadre de ce stage, il a été décidé de se concentrer sur l'outil Cheddar pour analyser un modèle de contrôleur central de réseau domotique.

3 Modélisation du contrôleur domotique avec AADL

Ce chapitre présente le travail de modélisation du contrôleur central du réseau domotique imaginé par Tréflévénet. Ce travail consiste à concevoir une architecture du contrôleur central qui répondra à l'ensemble des besoins fonctionnels et non fonctionnels définis.

Dans une première partie, nous présentons le système considéré et donnons l'ensemble des besoins fonctionnels et non fonctionnels attendus pour le contrôleur central. Dans une seconde partie, nous décrivons le travail de modélisation réalisé avec le langage AADL.

3.1 Description du système domotique

En ingénierie des systèmes, la conception d'un système démarre par la prise en compte des exigences fonctionnelles et non fonctionnelles. Les exigences fonctionnelles décrivent les fonctions que doit réaliser le système. Elles décrivent son comportement externe, et permettent au client et à l'ingénieur système de s'engager sur un périmètre et un ensemble de conditions à satisfaire. Les exigences non fonctionnelles sont toutes les spécifications qui n'expriment pas une fonction du logiciel. On distingue généralement deux types : les contraintes d'interface (environnement matériel, etc.) et les contraintes de performances (mémoire, disque, réactivité, etc.).

Cette section présente les spécifications du contrôleur central d'un réseau domotique. Dans une première partie, nous décrivons le réseau domotique imaginé par l'association Tréflévénet et nous positionnons le contrôleur central au sein de ce système. Puis nous détaillons les fonctionnalités que devra remplir ce contrôleur. Enfin, nous discutons des besoins non fonctionnels, et en particulier des contraintes temporelles à appliquer sur le contrôleur.

3.1.1 Réseau domotique considéré

Le réseau domotique est constitué d'un bus de terrain, d'un ensemble de modules esclaves, et d'un contrôleur central. Le contrôleur et chaque module sont connectés sur le bus de terrain à travers lequel ils échangent des messages. La figure 3.1 présente une vue simplifiée du réseau domotique.

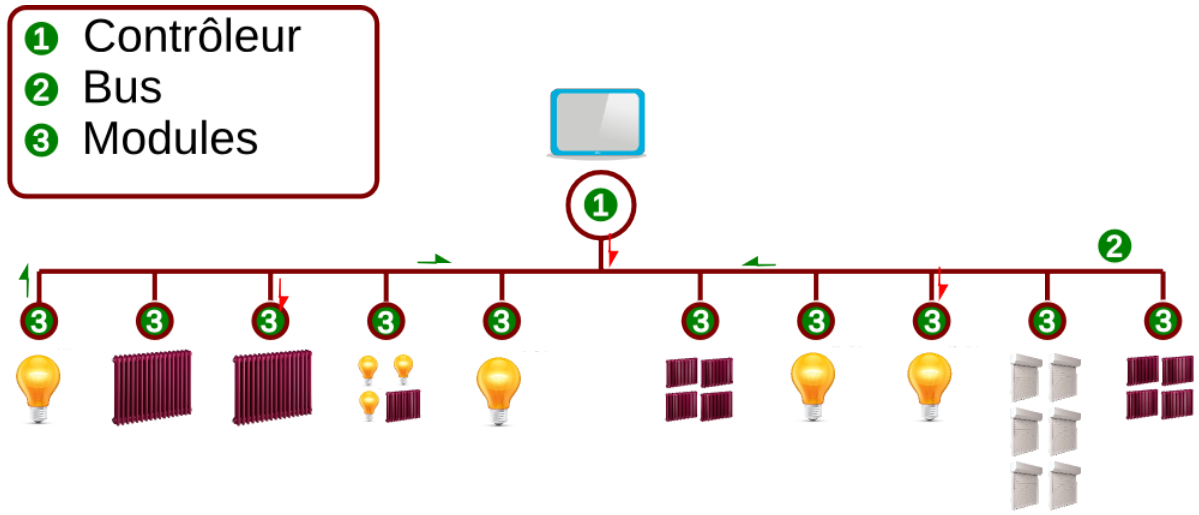

FIG. 3.1 – Architecture du système domotique

Le bus de terrain est le système d'interconnexion qui relie les modules esclaves au contrôleur central. Il repose sur une liaison série de type RS-485 configurée de 9600 bauds à 115200 bauds. Le tableau 3.1 présente les résultats de l'évaluation des débits et vitesses de transmission réalisée sur ce bus pour une configuration classique 8N1. Sachant que le protocole de communication spécifie une taille maximale de 20 octets pour les messages transitant sur le bus, on peut donc en déduire que le pire temps de propagation d'un message oscille entre 1.8 ms et 20.8 ms.

Débit global de la liaison (bauds)	Débit effectif (bits/s)	Délai de transmission d'un octet (us)
115200	92160	86.8
56600	46080	173.6
38400	30720	260.4
19200	15360	520.8
9600	7680	1 041.7

TAB. 3.1 – Performances du bus de terrain

Concernant les messages, leur format a été spécialement défini par Tréflévénet pour ce réseau domotique. Parmi les informations véhiculées, nous distinguons notamment :

- l'identifiant de l'émetteur du message ;
- l'identifiant du destinataire du message ;
- un checksum.

Les messages peuvent être émis soit en *unicast* (module vers contrôleur, ou contrôleur vers module), soit en *multicast* (contrôleur vers plusieurs modules). Tout message unicast doit être acquitté par son destinataire. Ainsi, si l'on ne considère pas le temps de traitement d'un message, la durée d'un échange de message unicast oscille entre 3.6 ms et 41.6 ms en fonction de la configuration du bus.

Les modules sont des nœuds d'entrées-sorties déportées raccordés au contrôleur central au moyen d'un bus de terrain. Ils apparaissent comme des équipements esclaves en

interaction avec l'équipement maître que constitue le contrôleur. L'architecture concernée s'apparente ainsi à un système distribué. Chaque module présente un certain nombre d'entrées et de sorties. Chaque entrée est configurable afin de prendre en compte des signaux analogiques (relevé de température, d'hygrométrie, etc.) ou numériques (lecture d'un bouton-poussoir, d'un capteur de fin de course, etc.). Selon les modules, les sorties sont basées sur l'utilisation de relais (contacts commutés) ou de triacs (contacts commutés ou variation de tension).

Le protocole de communication prévoie qu'un module qui émet un message à destination du contrôleur doit recevoir en retour un acquittement. Le temps limite de réception d'un acquittement a été fixé à 200 ms. Passé ce délai, le module considère que son message n'a pas été reçu par le contrôleur (plusieurs causes sont possibles : collision de messages, overflow côté contrôleur, etc.). Après un certain temps, il ré-émet son message. Le temps inter-émission est calculé par une fonction de génération de nombres pseudo-aléatoires qui se base sur l'adresse du module. De cette manière, il est possible d'éviter les émissions synchronisées de messages entre deux modules après un premier échec d'envoi.

Le contrôleur domotique est la partie centrale du système domotique. Il est lui aussi relié au bus de terrain, et joue le rôle de maître vis à vis des modules qu'il contrôle. Tout d'abord, le contrôleur est à l'écoute du bus de terrain pour recueillir les messages émis par les modules Chaque message lu entraîne la réalisation d'un ensemble de vérifications afin de s'assurer de l'intégrité du message :

- identification du module émetteur ;
- identification du destinataire du message ;
- calcul et comparaison du checksum du message.

Si le message est valide, le contrôleur émet un message d'acquittement à destination du module. Sinon, il ne fait rien. Le contrôleur peut émettre des commandes à destination d'un ou de plusieurs modules du réseau domotique. En mode unicast, tout message envoyé sur le bus devra être acquitté par le module destinataire, sinon ce message sera ré-émis après un délai calculé. En mode multicast, aucun acquittement n'est attendu.

Un opérateur peut agir sur le comportement du contrôleur au moyen d'une interface graphique affichée sur une dalle tactile directement reliée au contrôleur. Cette interface permet de visualiser l'état courant du réseau. Enfin, le contrôleur central peut réaliser des tâches pré-programmées.

3.1.2 Exigences fonctionnelles

Dans le cadre de cette étude, nous nous focalisons sur le contrôleur central du réseau domotique. Celui-ci doit satisfaire un ensemble de fonctionnalités que nous avons regroupées dans trois sous-groupes : 1) contrôle et supervision du bus de terrain, 2) gestion de l'interface graphique, et 3) gestion des tâches planifiées.

Le contrôle et la supervision du bus de terrain concerne toutes les fonctionnalités permettant d'écouter et d'interroger les modules esclaves du réseau domotique. Les échanges (requêtes et acquittements) sont réalisés sous forme de messages émis et reçus par le contrôleur via le bus de terrain. On distingue trois modes de communication : un mode de communication module esclave vers contrôleur (unicast), un mode de communication contrôleur vers un seul module esclave (unicast), et un mode de communication contrôleur vers un ensemble de modules esclaves (multicast).

Tout message lu doit être analysé afin de vérifier 1) que le module émetteur est bien connu du contrôleur, 2) que le contrôleur est effectivement le destinataire du message, et 3) que le contenu du message est correct (au moyen du calcul et de la comparaison d'un checksum). La prise en compte d'un message du bus peut entraîner des actions relatives au type de message. Si le message est un acquittement de commande envoyée précédemment, le contrôleur n'a plus à ré-émettre cette commande. Si le message indique le changement d'état d'un équipement électrique, le contrôleur traite ce changement d'état. Plusieurs actions peuvent être réalisées suite à la réception d'un message signifiant le changement d'état d'un équipement : envoi d'une commande à un ou plusieurs modules du réseau domotique, notification au niveau de l'interface graphique, annulation d'une tâche planifiée à l'avance, etc.

La gestion de l'interface graphique concerne toutes les interactions locales et distantes (via le web) avec l'utilisateur. Via la dalle tactile, il est possible de réaliser des commandes directes qui seront transmises au bus de terrain (commande d'ouverture/fermeture des volets, commande d'augmentation/diminution de la température des radiateurs, etc.). L'interface graphique permet d'afficher les données de supervision du réseau domotique (température des appareils de chauffage, état des volets, etc.). Elle offre la possibilité à l'utilisateur de configurer et de paramétrer le comportement du contrôleur central (planification de tâches, configuration des équipements, etc.). L'affichage de statistiques de fonctionnement des équipements électriques est aussi possible afin d'évaluer l'historique et l'évolution du réseau domotique.

La gestion des tâches planifiées permet de réaliser des commandes automatiques pour une date précise (volets roulants, arrosage, etc.). La granularité de définition des dates est la minute. Il est aussi possible de configurer un comportement à adopter en fonction de situations particulières. Par exemple, il sera possible d'adapter le niveau de marche des appareils de chauffage en fonction de la température extérieure prélevée par un capteur. La configuration du contrôleur se fait d'une part au démarrage à travers des fichiers de configuration, puis via une interface graphique accessible depuis la dalle tactile ou le web.

3.1.3 Exigences non fonctionnelles

Chacune des fonctionnalités du contrôleur décrites dans la partie précédente possède un niveau d'importance particulier. Ainsi, la fonctionnalité la plus critique est celle de contrôle et supervision du bus de terrain. Vient ensuite la fonctionnalité de gestion des tâches planifiées, puis la gestion de l'interface graphique.

Au niveau de la gestion du bus de terrain, le contrôleur doit assurer que tous les messages émis par les modules esclaves seront traités. La durée maximale pour le traitement d'un message est fonction de la configuration du bus, c'est-à-dire du temps de transfert du message. Pour rappel, les modules esclaves sont configurés de manière à considérer qu'un message émis a été perdu si aucun acquittement n'a été reçu 200 ms après l'envoi du message. Par conséquent, le temps de transition du message à travers le bus aura une incidence sur le temps maximal de traitement par le contrôleur. Par exemple, dans le cas où le bus de terrain est configuré pour fournir un débit effectif de 7680 bits/s, la durée de transfert au pire cas d'un message de 20 octets est donc de 21 ms. Par conséquent, le temps de transfert du message et de son acquittement nécessite 42 ms, ce qui laisse 158 ms au contrôleur pour le traitement.

Le bus de terrain dispose d'un buffer de 128 octets. Cela signifie qu'il peut contenir au maximum 6 messages de modules. Le contrôleur doit s'assurer que ce buffer n'est jamais plein afin d'éviter la perte de messages. Cette fois encore, la durée entre deux dépilements de message est fonction de la configuration du bus de terrain. Par exemple, dans une configuration avec un débit effectif de 7680 bits/s, un message est potentiellement déposé dans le buffer toutes les 21 ms.

Chaque message lu depuis le bus doit être traité dans un intervalle de 150 ms. Une action ihm doit être traitée le temps d'un battement de paupière (environ 150 ms).

Concernant les contraintes matérielles, le contrôleur central sera déployé sur une carte électronique. Cette carte doit notamment disposer d'un connecteur RS-485 pour relier le bus de terrain, et d'un processeur monocore pour exécuter la partie applicative.

3.2 Présentation du modèle de contrôleur domotique

AADL est le langage de description d'architecture qui a été choisi pour modéliser le contrôleur central du réseau domotique. Comme nous l'avons expliqué dans la section 2.2, AADL propose un ensemble de fonctionnalités très intéressantes, notamment 1) la possibilité de différencier la partie matérielle de la partie logicielle d'un système, 2) la possibilité de créer plusieurs vues d'une même architecture, et 3) la présence d'un grand nombre d'outils pour réaliser différentes analyses. Cheddar est un de ces outils. Il est notamment utilisé pour réaliser des études d'ordonnançabilité sur des systèmes.

Dans cette partie, nous présentons le modèle du contrôleur central du réseau domotique réalisé à l'aide d'AADL. Dans un premier temps, nous donnons une vue haut-niveau du contrôleur central au sein du réseau domotique. Puis nous descendons à l'intérieur du contrôleur afin de déterminer ses composants internes matériels et logiciels.

3.2.1 Modélisation du réseau domotique

La figure 3.1 représente le réseau domotique considéré. On distingue principalement trois composants : 1) le contrôleur central, 2) le bus de terrain par lequel transitent les messages, et 3) les modules esclaves. Le premier travail de modélisation consiste à représenter ce système grâce aux éléments de langage mis à disposition par AADL.

Le contrôleur central est pour l'instant représenté par un composant de type système. Pour rappel, il s'agit d'un élément composite du langage permettant de représenter une entité à l'intérieur de laquelle il est possible de zoomer pour spécifier ses composants internes. Le bus de terrain est représenté quant à lui par un élément de type bus. Enfin, les modules sont représentés par un élément de type device. Le concept du device est similaire à celui de système, sauf que cette fois-ci l'entité est considérée comme une boite noire. En effet, nous ne cherchons pas à savoir de quoi sont constitués les modules. Nous avons juste besoin de connaître leur interface afin de pouvoir la connecter à celle du contrôleur.

Le contrôleur central communique avec les modules via des messages transitant par le bus de terrain. Sous AADL, les interfaces de composants permettent de décrire les signaux entrants et sortants des composants. On distingue les signaux pouvant être bufferisés dans

une file de type FIFO (**event port**, **event data port**) des signaux non bufferisés (**data port**). Dans notre cas, le bus de terrain dispose d'un buffer de réception, les signaux utilisés seront donc échangés entre des ports de type **event data port**. Il est possible de regrouper un ensemble de ports au sein d'un groupe de ports appelé **feature group** (**port group** dans la première version d'AADL). Ce type d'entité permet de considérer un ensemble de signaux en une seule fois, et surtout de créer des ports *miroirs* au moyen du mot-clé **inverse of**. L'extrait de code 3.1 présente la déclaration des ports d'entrée et de sortie du contrôleur central, et la déclaration des ports miroirs associés.

```
feature group dcs_socket
features
    -- signal de module
    sensor_signal: in event data port bus_message.module_signal;
    -- acquittement de signal
    sensor_signal_ack: out event data port bus_message.module_signal_ack;

    -- message de commande
    actuator_command: out event data port bus_message.module_command;
     -- acquittement de commande
    actuator_command_ack: in event data port bus_message.module_command_ack;
end dcs_socket;

feature group dcs_plug inverse of dcs_socket
end dcs_plug;
```

Src. 3.1 – Déclaration des groupes de ports de signaux

Dans ce groupe de ports, nous avons défini 4 ports :
- un port `sensor_signal` permettant de recevoir les évènements émis par les modules à destination du contrôleur ;
- un port `sensor_signal_ack` par lequel sont émis les acquittements de signaux par le contrôleur à destination du module ;
- un port `actuator_command` par lequel sont émisent les commandes par le contrôleur à destination d'un ou de plusieurs modules ;
- un port `actuator_command_ack` permettant de recevoir les acquittements de commande émis par les modules à destination du contrôleur.

Tous les messages échangés sont du type `bus_message`. Il s'agit d'une entité AADL de type data proposant 4 implantations possibles : `module_signal`, `module_signal_ack`, `module_command`, et `module_command_ack`. Le typage des ports permet notamment de repérer rapidement les erreurs de connexion de ports durant l'étape de modélisation, et consolide le modèle.

Le résultat de ce premier travail de modélisation est disponible à l'annexe A. La représentation graphique du système ainsi modélisé est visible sur la figure 3.2. Cette première représentation permet de bien distinguer les différentes entités. Le contrôleur central est constitué d'un processus `domotx_appli` et d'un composant de type système `domotx_board` permettant de représenter la carte électronique sur laquelle sera déployé le processus. En face, on distingue un module représenté par le device `domotx_module`. Le processus du contrôleur et le module sont reliés par le feature group déclaré au niveau de leur interface. Enfin, le module et la partie matérielle du contrôleur sont interconnectés via le bus de terrain `domotx_bus` pour le transfert des messages.

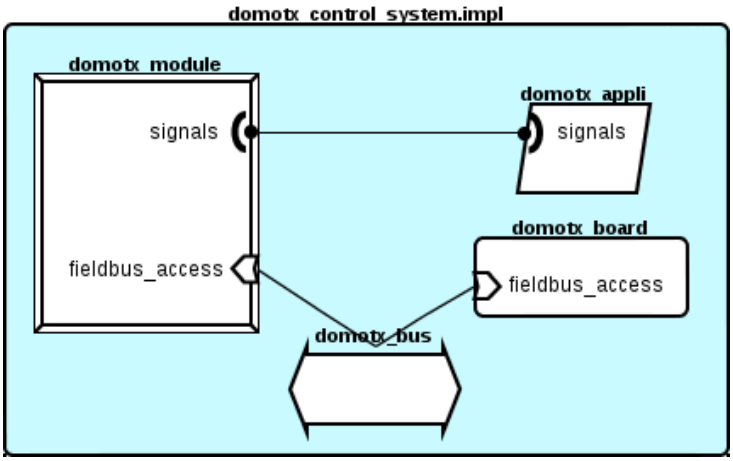

FIG. 3.2 – Vue de haut niveau du système

3.2.2 Modélisation du contrôleur central

Durant le premier travail de modélisation du réseau domotique, nous avons choisi de représenter le contrôleur central en deux parties. Tout d'abord une partie matérielle avec un composant de type system appelé `dcs_board`. Puis une partie logicielle avec l'utilisation d'un composant de type process appelé `dcs_appli`. Cette partie décrit en détail l'étape de modélisation du contrôleur central du réseau domotique. Dans un premier temps, nous présentons la modélisation de la partie matérielle, puis nous continuons par la modélisation de la partie applicative.

Afin de permettre l'exécution d'un processus, il est nécessaire de disposer d'un support physique d'exécution, et notamment d'un processeur et d'une mémoire. Pour rappel, une des exigences non fonctionnelles concernant le contrôleur central est que celui-ci sera déployé sur une carte électronique dont les caractéristiques exactes n'ont pas encore été entièrement figées. Nous savons cependant que la carte disposera d'un processeur monocore et qu'elle devra être reliée au bus de terrain.

Même si nous ne disposons pas de toutes les caractéristiques de la carte électronique qui servira de support, il est quand même possible de démarrer sa modélisation. AADL permet en effet, au moyen des interfaces de composant, de déclarer des entités qu'il est possible de raffiner par la suite. L'objectif de la modélisation de la carte électronique est donc de décrire un modèle suffisamment général pour pouvoir être adapté à tout type de carte, et en même temps suffisamment complet pour répondre à notre problématique. Ici, nous avons besoin de deux entités : un processeur et une mémoire associée.

Nous déclarons un composant de type processor appelé `board_processor`. Ce processeur doit avoir accès à deux bus : 1) le bus de terrain afin de lire et écrire les messages, et 2) un bus interne permettant de relier le processeur à sa mémoire. Pour déclarer qu'un composant nécessite l'accès à un bus, nous utilisons le mot-clé **requires bus access** dans l'interface. Il est possible de préciser des propriétés afin d'améliorer la description du processeur. Nous pouvons notamment citer **Scheduling_Protocol**, **Preemptive_Scheduler**, et **Scheduler_Quantum** sur lesquelles nous reviendrons dans le chapitre suivant. L'extrait de code 3.2 présente la déclaration du processeur de la carte électronique.

```
-- déclaration du processeur
processor board_processor
features
    fieldbus_access: requires bus access domotx_fieldbus;
    cpubus_access  : requires bus access board_cpubus;
properties
    Scheduling_Protocol  => (POSIX_1003_Highest_Priority_First_Protocol);
    Preemptive_Scheduler => True;
    Scheduler_Quantum    => 1;
end board_processor;
```

Src. 3.2 – Déclaration du processeur de la carte

Ce processeur accède à sa mémoire via le bus `cpubus_access`. Nous déclarons donc un composant de type memory appelé `board_memory` pour représenter cette mémoire. Dans le cadre de notre étude, il n'est pas nécessaire de préciser les propriétés de ce composant. Pour finir, nous associons le processeur et sa mémoire au niveau de l'implantation du processeur appelée `board_processor.with_memory`. L'extrait de code 3.3 présente la déclaration du composant mémoire, et son intégration en tant que sous-composant du processeur de la carte électronique.

```
-- déclaration de la mémoire
memory board_ram
end board_ram;

processor implementation board_processor.impl
subcomponents
    ram: memory board_ram;
end board_processor.imp;
```

Src. 3.3 – Implantation du processeur de la carte

Nous venons de déclarer les principaux composants que devra contenir la carte électronique du contrôleur central. Nous pouvons à présent implanter l'interface `dcs_board` que nous avions définie dans la première étape de modélisation de la carte électronique. Comme il est nécessaire de pouvoir accéder au bus de terrain, nous créons la connexion nécessaire afin de relier le processeur de la carte au bus. L'extrait de code 3.4 présente la déclaration complète du composant `dcs_board` qui modélise la partie matérielle de la carte électronique. La figure 3.4 permet de visualiser les connexions du processeur sur les bus.

```
system dcs_board
features
    fieldbus_access: requires bus access domotx_fieldbus;
end dcs_board;

system implementation dcs_board.impl
subcomponents
    board_cpu: processor board_processor.with_memory;
    board_bus: bus board_cpubus.impl;
connections
    -- connexions au bus
    B1: bus access fieldbus_access -> board_cpu.fieldbus_access;
    B2: bus access board_bus -> board_cpu.cpubus_access;
end dcs_board.impl;
```

Src. 3.4 – Déclaration de la carte électronique

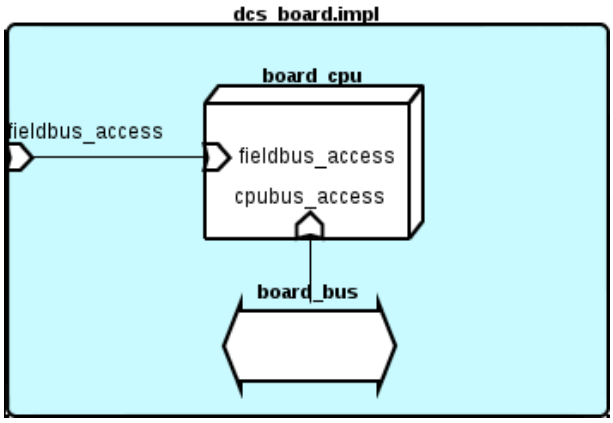

FIG. 3.3 – Représentation graphique de la partie matérielle

Un processus `dcs_appli` a été défini pour représenter la partie applicative du contrôleur central. Pour ce processus, nous avons défini un ensemble de ports regroupés dans un feature group appelé `dcs_socket`. Ces ports représentent les échanges que le processus réalise avec l'extérieur, c'est-à-dire le reste du réseau domotique. Le travail de modélisation de la partie logicielle du contrôleur consiste à préciser le contenu du processus `dcs_appli` en déterminant ses sous-composants et leurs connexions.

Le processus doit au minimum réaliser trois tâches : 1) contrôler et superviser le bus de terrain (lire et écrire les messages sur le bus), 2) gérer l'interface graphique, et 3) gérer les tâches planifiées. Généralement, les tâches sont représentées sous AADL avec des composants de type **thread**. A ce stade de la modélisation, il existe plusieurs moyens de concevoir le fonctionnement interne du processus via les threads. Par exemple, nous pouvons considérer que l'ensemble des tâches sera réalisé par un seul thread chargé d'effectuer l'ensemble du travail. Ou nous pouvons associer à chaque tâche un thread, ce qui ferait trois threads à modéliser pour décrire le fonctionnement du processus.

Cependant, afin de prendre en considération les besoins non fonctionnels du système en terme de criticité des échanges avec les modules esclaves, nous avons décidé d'utiliser cinq threads : un thread chargé de lire les messages sur le bus, un thread chargé d'écrire des messages sur le bus, un thread chargé des tâches planifiées, un thread chargé de la gestion de l'interface graphique, et un thread chargé des traitements internes.

Pour le thread chargé de lire les messages sur le bus, nous avons déclaré un composant de type thread appelé `fieldbus_reader`. Au niveau de son interface, nous déclarons deux event data ports en entrée `signal` et `command_ack` pour lire respectivement les signalisations et les acquittements de commande des modules esclaves. Ce thread doit vérifier chaque message lu (identification de l'émetteur et du destinataire, calcul du checksum). Toute signalisation vérifiée entraînera l'émission d'une notification d'envoi d'acquittement à destination du thread chargé d'écrire les messages sur le bus. Tout message vérifié sera notifié au thread chargé des traitements internes. Les notifications à destination d'autres threads passent à travers un event data port en sortie appelé `notification_output`. Pour représenter les notifications internes au processus du contrôleur central, nous avons déclaré un type de donnée `domotx_message`. Enfin, comme le thread `fieldbus_reader` a besoin d'accéder au bus de terrain, tout comme le thread chargé d'écrire sur ce bus, nous avons besoin d'indiquer un accès exclusif sur cette ressource. Pour cela, nous déclarons un composant de type data appelé `domotx_shared`. Cette entité décrit un verrou

sur une ressource. Au niveau de la déclaration du thread, nous précisons l'accès exclusif à cette data au moyen du mot-clé **requires data access**. Dans le chapitre suivant, nous reviendrons sur la notion d'accès exclusif à une ressource. L'extrait de code 3.5 présente la déclaration du thread chargé de lire les messages depuis le bus de terrain.

```
-- déclaration du thread chargé de lire sur le bus de terrain les messages
-- émis par les modules, et de retourner un acquittement pour les signaux
-- corrects
thread fieldbus_reader
features
    -- signal de module
    signal: in event data port bus_message.module_signal;
    -- acquittement de commande
    command_ack: in  event data port bus_message.module_command_ack;

    -- émission de notifications vers les autres managers
    notification_output: out event data port domotx_message;

    -- accès exclusif sur le bus de terrain
    fieldbus_lock: requires data access domotx_shared.fieldbus;
end fieldbus_reader;
```

Src. 3.5 – Déclaration du thread *fieldbus_reader*

Pour le thread chargé d'écrire les messages de commande et d'acquittement sur le bus, nous avons utilisé un composant de type thread appelé `fieldbus_writer`. Au niveau de son interface, nous déclarons deux event data ports en sortie `command` et `signal_ack` pour envoyer respectivement les commandes et les acquittements de signaux. Ce thread est seulement chargé d'écrire sur le bus les messages qui lui sont notifiés par les autres threads du processus. Pour recevoir les notifications d'émission, nous déclarons un event data port en entrée appelé `notification_input`. Enfin, pour représenter l'accès exclusif à la ressource bus de terrain, nous utilisons cette fois encore le mot-clé **requires data access** sur l'instance de `domotx_shared`. L'extrait du code 3.6 présente la déclaration du thread chargé d'écrire les messages sur le bus de terrain.

```
-- déclaration du thread chargé d'envoyer des commandes à destination d'un
-- ou de plusieurs modules
thread fieldbus_writer
features
    -- commande pour un ou plusieurs modules
    command: out event data port bus_message.module_command;
    -- acquittement de signal
    signal_ack: out event data port bus_message.module_signal_ack;

    -- réception des notifications émises par d'autres managers
    notification_input: in event data port domotx_message;

    -- accès exclusif sur le bus de terrain
    fieldbus_lock: requires data access domotx_shared.fieldbus;
end fieldbus_writer;
```

Src. 3.6 – Déclaration du thread *fieldbus_writer*

Le thread chargé de la gestion des tâches planifiées est appelé `task_scheduler`. Son rôle consiste à vérifier toutes les minutes si une tâche planifiée doit être réalisée par le contrôleur. Si c'est le cas, il envoie une notification à destination du thread chargé des traitements internes. Pour représenter cela, nous déclarons dans l'interface du thread un event data port en sortie appelé `notification_output`. L'extrait de code 3.7 présente la déclaration du thread chargé de la gestion des tâches planifiées.

```
-- déclaration du thread chargé d'envoyer des notifications de tâches à réaliser
-- lorsque le moment est venu
thread task_scheduler
features
    -- message à traiter par le core manager
    notification_output: out event data port domotx_message;
end task_scheduler;
```

Src. 3.7 – Déclaration du thread *task_scheduler*

Le thread chargé de la gestion de l'interface graphique est appelé `gui_controller`. Son rôle consiste à traiter les évènements IHM d'une part, et à mettre à jour l'affichage dès que nécessaire. Pour décrire cela, nous déclarons deux event data ports, l'un en entrée et l'autre en sortie. Le port est entrée est appelé `event_input`. Il permet de recevoir dans une file FIFO les évènements de type clic de souris. Le thread `gui_controller` est capable d'interpréter ces évènements et de les transmettre au thread chargé des traitements internes au moyen de notifications. Pour transmettre ces notifications, nous déclarons un event data port en sortie appelé `notification_output`. Le port en sortie est appelé `refresh_output`. Il permet de transférer les mises à jour de l'IHM afin de rafraîchir l'information affichée à l'utilisateur. Le thread `gui_controller` reçoit ses commandes de mise à jour à partir d'une file FIFO représentée par un event data port en entrée appelé `notification_input`. L'extrait de code 3.8 présente la déclaration du thread chargé de la gestion de l'interface graphique.

```
-- déclaration du thread chargé des interactions avec l'interface graphique
thread gui_controller
features
    -- prise en compte évènement ihm
    event_input: in event data port gui_event;
    -- mise à jour de l'affichage
    refresh_output: out event data port domotx_message;

    -- réception des notifications émises par d'autres managers
    notification_input: in event data port domotx_message;
    -- émission de notifications vers les autres managers
    notification_output: out event data port domotx_message;
end gui_controller;
```

Src. 3.8 – Déclaration du thread *gui_controller*

Enfin le thread chargé des traitements internes est déclaré via un composant de type thread appelé `core_manager`. Il a deux objectifs : 1) traiter les notifications qui lui sont envoyées par les autres threads du processus, 2) réaliser les traitements particuliers. Pour récupérer les notifications des autres threads, nous déclarons un event data port en entrée appelé `notification_input`. C'est depuis cette file FIFO que `core_manager` récupère les notifications des threads `fieldbus_reader`, `task_scheduler`, et `gui_controller`. Chaque message récupéré peut faire l'objet d'un traitement particulier. Ce traitement est configuré au démarrage du contrôleur, mais peut aussi être modifié par l'utilisateur au travers de l'interface graphique. Quoiqu'il en soit, au terme du traitement, le thread est chargé de notifier les différents managers des actions à réaliser. Cela peut être une mise à jour de l'interface graphique, ou bien l'émission d'un ou de plusieurs messages sur le bus de terrain. Pour cela, nous déclarons un event data port en sortie appelé `notification_output`. Le thread `core_manager` est aussi chargé de s'assurer de la réception des acquittements pour les commandes envoyées à des modules en mode unicast. L'absence de notification d'acquittement reçue par le thread *fieldbus_reader* fait l'objet d'un traitement particulier correspondant à la ré-émission de la commande à destination

du module qui n'a pas répondu. L'extrait de code 3.9 présente la déclaration du thread chargé des traitements internes du contrôleur.

```
-- déclaration du thread principal chargé de traiter les notifications émises
-- par les autres tâches
thread core_manager
features
    -- réception des notifications émises par d'autres managers
    notification_input : in  event data port domotx_message;
    -- émission de notifications vers les autres managers
    notification_output: out event data port domotx_message;
end core_manager;
```

Src. 3.9 – Déclaration du thread *core_manager*

Finalement, nous complétons la déclaration du processus `dcs_appli` au niveau de son implantation afin de déclarer les différents threads qui le composent. C'est aussi à ce niveau du modèle que nous déclarons les connexions qui relient les threads du processus à l'interface. Ainsi, le thread `fieldbus_reader`, chargé de lire les messages du bus, est connecté aux ports entrants du processus `sensor_signal` et `actuator_command_ack`. De manière analogue, le thread `fieldbus_writer`, chargé d'écrire les messages sur le bus, est connecté aux ports sortants du processus `signal_ack` et `command`. Pour le verrou sur le bus de terrain, nous déclarons dans notre processus un sous-composant `bus_lock` de type `domotx_shared`. Nous donnons un accès à ce verrou aux threads `fieldbus_reader` et `fieldbus_writer` dans la section réservée aux connexions des sous-composants avec le mot-clé **data access**. Enfin, nous déclarons les connexions inter-threads dans la section réservée aux connexions. Le thread `fieldbus_reader` peut émettre des notifications à destination de `core_manager` et `fieldbus_writer`. Les threads `task_scheduler` et `gui_controller` peuvent aussi émettre des notifications à destination de `core_manager`. Enfin, le thread `core_manager` peut émettre des notifications à destination des threads `fieldbus_writer` et `gui_controller`. L'extrait de code 3.10 présente la déclaration complète du processus *dcs_appli* qui modélise la partie logicielle du contrôleur. La figure 3.4 permet de mieux visualiser les connexions internes et les relations entre les threads.

```
process dcs_appli
features
    signals: feature group dcs_socket;
end dcs_appli;

process implementation dcs_appli.impl
subcomponents
    reader   : thread fieldbus_reader;
    writer   : thread fieldbus_writer;
    gui      : thread gui_controller;
    scheduler: thread task_scheduler;
    core     : thread core_manager;

    bus_lock : data domotx_shared.fieldbus;

connections
    -- connexions avec l'extérieur
    C_sensor_reader: port signals.sensor_signal -> reader.signal;
    C_actuator_reader: port signals.actuator_command_ack -> reader.command_ack;
    C_writer_sensor: port writer.signal_ack -> signals.sensor_signal_ack;
    C_writer_actuator: port writer.command -> signals.actuator_command;

    -- connexions internes au contrôleur
    C_reader_core: port reader.notification_output -> core.notification_input;
    C_reader_writer: port reader.notification_output -> writer.notification_input;
    C_gui_core: port gui.notification_output -> core.notification_input;
    C_scheduler_core: port scheduler.notification_output -> core.notification_input;
    C_core_gui: port core.notification_output -> gui.notification_input;
```

```
    C_core_writer: port core.notification_output -> writer.notification_input;

    -- accès exclusif au bus de terrain
    D_fieldbus_1: data access bus_lock -> reader.fieldbus_lock;
    D_fieldbus_2: data access bus_lock -> writer.fieldbus_lock;

end dcs_appli.impl;
```

Src. 3.10 – Déclaration du processus *dcs_appli*

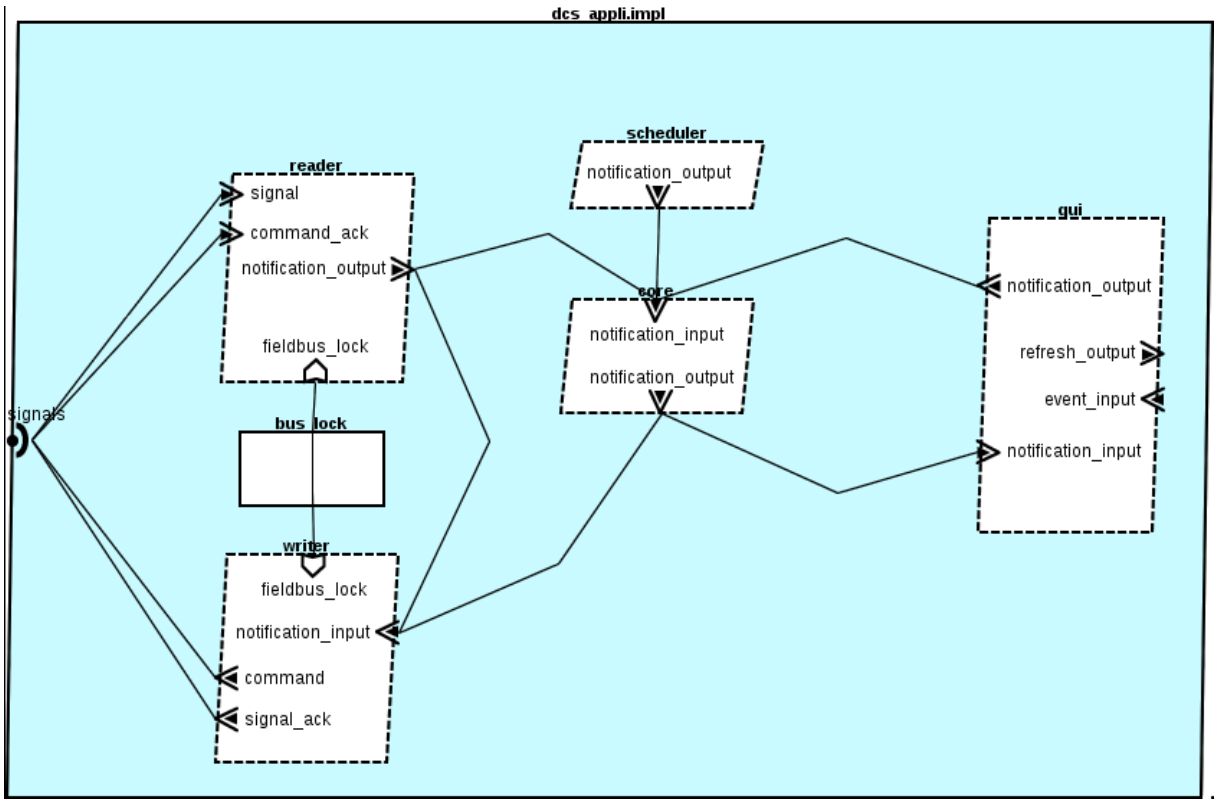

Fig. 3.4 – Représentation graphique de la partie logicielle

3.3 Conclusion

Dans ce chapitre, nous avons montré comment il est possible de modéliser une architecture avec AADL à partir de la prise en compte des exigences fonctionnelles et non fonctionnelles d'un système. La première étape de la modélisation à consister à représenter le contrôleur central au sein du réseau domotique. Ainsi, nous avons pu déterminer les messages échangés à travers le bus de terrain, et nous avons aussi défini l'interface externe du contrôleur. Celui-ci se décompose en deux parties : une partie matérielle et une partie logicielle.

La partie matérielle permet de décrire le support d'exécution de la partie logicielle. Etant donné que le choix de la carte électronique n'a pas encore été fixé, nous ne disposons pas de toutes les informations permettant de la décrire. Nous avons pu cependant déterminer les composants nécessaires à notre travail d'étude : le processeur (qui est relié au bus de terrain) et sa mémoire. Ces deux composants sont suffisants pour réaliser les analyses décrites dans le chapitre suivant. Cette partie pourra être raffinée dans un second

temps : comme partie matérielle et applicative sont distinctes, cette modification n'aura, a priori, que peu d'impacts sur la partie logicielle.

La partie logicielle permet de décrire le comportement du système. Pour cela, nous avons déclaré un ensemble de threads et leurs connexions. Chaque thread accompli une tâche particulière et communique avec un ou plusieurs threads au moyen de notifications. Nous avons représenté le fait que deux threads se partagent exclusivement la ressource bus de terrain. Dans le chapitre suivant, nous allons voir comment ces threads sont implantés afin de réaliser notre étude qualificative du contrôleur central.

4 Analyse de performances temps-réel avec AADL

Ce chapitre présente l'étape d'analyse des performances réalisée sur le modèle AADL du contrôleur central du réseau domotique. Ce travail a consisté à établir les propriétés du jeu de tâches à partir d'un certain nombre de conjectures, puis à utiliser le framework AADLInspector pour réaliser des tests de faisabilité et des simulations d'ordonnancement.

Après avoir décrit la méthodologie globale de test, nous présentons les hypothèses de départ qui nous ont permis d'établir notre jeu de tâches. Puis, nous expliquons comment nous avons implanté ce jeu de tâches dans notre modèle AADL pour réaliser les tests et les simulations avec AADLInspector. Enfin, nous présentons les résultats obtenus.

4.1 Méthodologie globale

Pour rappel, le sujet du stage porte sur la qualification en amont d'un système décrit avec le langage de description d'architecture AADL. L'idée est de vérifier s'il est possible d'évaluer qu'un système est ordonnançable sans toutefois avoir écrit une seule ligne de code. C'est la raison pour laquelle nous seront amenés à faire un certain nombre de conjecture sur le système considéré.

La méthodologie que nous proposons pour vérifier le respect des contraintes temporelles se base les outils AADLInspector et Cheddar v3. A partir du jeu de tâches que nous avons défini dans le chapitre précédent, nous allons déterminer les propriétés d'ordonnancement à partir d'un ensemble de conjectures de départ sur le comportement attendu du contrôleur central. La capacité de chaque tâche sera évaluée en prenant en compte tout d'abord les traitements réalisés au pire cas. Par exemple, la capacité de `fieldbus_reader` sera fonction du nombre de messages lus sur le bus de terrain et du nombre de notifications émises à destination de `fieldbus_writer` et `core_manager`. Pour chaque tâche, un travail d'évaluation de son algorithme sera réalisé et permettra d'estimer les autres caractéristiques du jeu de tâches. Nous implanterons ensuite notre modèle sous AADLInspector. Ce framework édité par Ellidiss permet notamment de réaliser des tests de faisabilité et des simulations d'ordonnancement en se basant sur le moteur de Cheddar v3. Nous analyserons alors les résultats obtenus afin de valider ou non la configuration du jeu de tâches.

4.2 Estimation des caractéristiques du jeu de tâches

Le jeu de tâches est constitué de cinq tâches. Chaque tâche a une importance relative à la fonctionnalité qu'elle assure. Pour rappel, le contrôleur doit assurer principalement trois fonctionnalités qui sont dans leur ordre d'importance : le contrôle et la supervision du bus de terrain, la gestion des tâches planifiées, et la gestion de l'interface graphique.

Cette section présente l'estimation des caractéristiques d'ordonnancement des différentes tâches du contrôleur. Dans une première partie, nous calculons la capacité C de chaque tâche. Dans une seconde partie, nous évaluons les autres caractéristiques d'ordonnancement en fonction de la configuration du bus de terrain.

4.2.1 Détermination des capacités des tâches

Pour déterminer la capacité de chaque tâche, nous allons d'abord présenter leur algorithme. Celui-ci décrit les opérations réalisées à chaque déclenchement de la tâche. Puis, nous ferons des hypothèses sur les conditions de déroulement des tests.

`fieldbus_reader` est le thread chargé de lire les messages arrivant au contrôleur par le bus de terrain. Pour chaque message lu, un ensemble de vérifications est réalisé afin de déterminer sa validité. En fonction de la nature du message, le thread peut notifier le thread `fieldbus_writer` pour demander l'envoi d'un acquittement à l'émetteur du message. Dans tous les cas, tout message correct lu par `fieldbus_reader` est transmis au `core_manager` pour traitement. L'algorithme en pseudo-code du thread `fieldbus_reader` est le suivant

```
1  tantque il reste un message à dépiler du buffer du bus de terrain faire
2      dépiler un nouveau message
3      vérifier la validité du message (émetteur, destinataire, checksum)
4      si le message est correct alors
5          si le message est un signal de module alors
6              notifier fieldbus_writer d'envoyer un acquittement
7          fin si
8          notifier core_manager de la réception d'un nouveau message
9      fin si
10 fin tantque
```

Nous estimons que la vérification de la présence d'un message et son dépilement sont des opérations atomiques pouvant être considérées comme immédiates. La vérification du message lu consiste en un ensemble de tests dont la durée totale au pire cas (message correct) est estimée à 2 ms. Enfin, nous convenons que le temps pris pour notifier les autres threads est une opération suffisamment courte pour être considérée comme immédiate. Par conséquent, si on note m le nombre de messages présents dans le buffer du bus de terrain au moment où le thread est déclenché, alors le temps d'exécution sera $2m$ ms.

`fieldbus_writer` est le thread chargé d'écrire les messages à destination des modules sur le bus de terrain. Pour chaque notification de demande d'émission de message provenant de `fieldbus_reader` ou de `core_manager`, un message sera écrit sur le bus. L'algorithme en pseudo-code du thread `fieldbus_writer` est le suivant

```
1  tantque la file des notifications n'est pas vide faire
2      dépiler une nouvelle notification
3      émettre le message contenu dans la notification sur le bus
4  fin tantque
```

Nous estimons que la vérification de la présence d'une notification et son dépilement sont des opérations atomiques pouvant être considérées comme immédiates. L'opération d'écriture sur le bus est estimée à 1 ms par message. Par conséquent, si on note n le nombre de notifications présentes dans la file de `fieldbus_writer` lors de son déclenchement, alors le temps d'exécution sera n ms.

`task_scheduler` vérifie chaque minute si une tâche planifiée doit être réalisée par le contrôleur. Si tel est le cas, alors une notification est émise à destination de `core_manager` qui sera alors chargé de réaliser le traitement adéquat. L'algorithme en pseudo-code du thread `task_scheduler` est le suivant

```
1 tantque une action planifiée est à réaliser faire
2     notifier core_manager de l'action à réaliser
3 fin tantque
```

Même si les opérations de vérification de l'existence d'actions planifiées et d'envoi de notifications vers `core_manager` peuvent être considérées comme atomiques, nous convenons que le temps d'exécution de `task_scheduler` sera de 1 ms à chaque déclenchement.

`gui_controller` réalise deux opérations. Tout d'abord, il est chargé de traiter les évènements IHM et de les transmettre à `core_manager` pour traitement. Sa seconde mission consiste à mettre à jour l'interface graphique à partir des informations remontées par `core_manager`. L'algorithme en pseudo-code du thread `gui_controller` est le suivant

```
1 tantque un évènement IHM est à traiter faire
2     dépiler un évènement IHM
3     notifier core_manager d'une action à réaliser
4 fin tantque
5 tantque une notification est à traiter faire
6     dépiler une demande de rafraichissement de l'interface graphique
7 fin tantque
8 rafraîchir l'interface graphique
```

Même si les opérations de vérification de l'existence d'un évènement IHM et d'envoi de notifications vers `core_manager` peuvent être considérées comme atomiques, nous convenons que le temps de traitement des évènements IHM sera de 1 ms. Nous faisons de même pour la vérification et le dépilement de demande de mise à jour de l'interface graphique. Par contre, les opérations IHM étant relativement plus coûteuses en temps CPU, nous convenons que la durée d'un rafraichissement de l'interface graphique sera de 8 ms. Par conséquent, le temps d'exécution de `gui_controller` sera de 10 ms.

`core_manager` a pour rôle de réaliser le traitement des notifications qu'il reçoit de `fieldbus_reader`, `task_scheduler`, et `gui_controller`. Chaque notification peut avoir un ou plusieurs résultats :

- mise à jour du modèle de données interne du manager ;
- envoi d'un ou de plusieurs messages sur le réseau ;
- mise à jour de l'interface graphique.

L'algorithme en pseudo-code du thread `core_manager` est le suivant

```
1  tantque la file des notifications n'est pas vide faire
2      dépiler une nouvelle notification
3      traiter la notification
4      si le traitement nécessite l'émission de messages alors
5          notifier fieldbus_writer des messages à émettre
6      fin si
7      si le traitement nécessite une mise à jour de l'interface graphique alors
8          notifier gui_controller de rafraichir l'IHM
9      fin si
10 fin tantque
```

Nous estimons que la vérification de la présence d'une notification et son dépilement sont des opérations atomiques pouvant être considérées comme immédiates. Le traitement d'une notification est estimé à 1 ms dans le pire des cas. Enfin, les opérations de notification de `fieldbus_writer` et `gui_controller` sont évaluées à 1 ms en tout. Par conséquent, si on note n le nombre de notifications présentes dans la file de `core_manager`, alors le temps d'exécution sera $2n$ ms.

4.2.2 Caractérisation du jeu de tâches

Pour réaliser des tests d'ordonnancement, il est nécessaire de renseigner un ensemble de propriétés :

- S_i : la date d'arrivée de la tâche i dans le système ;
- C_i : la capacité de la tâche i ;
- P_i : la période d'activation de la tâche i ;
- D_i : le délai critique (ou *deadline*) de la tâche i exprimé relativement à P_i.

Nous avons vu que la capacité de la tâche `fieldbus_reader` dépend du nombre de messages à lire sur le bus de terrain. Les capacités des tâches `fieldbus_writer` et `core_manager` dépendent quant à elles du nombre de notifications reçues. Cette partie présente d'abord les suppositions que nous avons faites afin de simplifier le modèle du jeu de tâches. Puis nous décrivons la configuration que nous avons sélectionnée pour réaliser les tests d'ordonnançabilité.

Nous savons que la configuration du bus influe sur le nombre de messages arrivant au niveau du contrôleur pour un intervalle de temps donné. De plus, nous savons que le bus de terrain dispose d'un buffer d'une taille de 128 octets. Comme le protocole de communication prévoit une taille de message maximale de 20 octets, alors le buffer peut contenir au mieux jusqu'à 6 messages. Comme nous ne souhaitons perdre aucun message qui est envoyé au contrôleur, dans le pire cas le contrôleur sera amené à traiter 6 messages. Puisqu'il faut 2 ms pour traiter un message au niveau du thread `fieldbus_reader`, alors sa capacité sera comprise sur un intervalle de 0 ms (cas où il n'y a aucun message) à 12 ms (cas où il y a 6 messages corrects).

Afin de faciliter le modèle pour estimer la capacité des threads `fieldbus_writer` et `core_manager`, nous allons faire les suppositions suivantes :

- tout message lu sur le bus aura pour traitement l'envoi d'un seul message unicast (ou multicast) sur le bus de terrain ;
- chaque minute, une action préprogrammée sera à réaliser, et elle se traduira par l'envoi d'un seul message unicast sur le bus de terrain ;
- à chaque déclenchement de `gui_controller`, un évènement IHM sera à traiter, et il se traduira par l'envoi d'un seul message unicast sur le bus de terrain ;
- tous les messages émis par le contrôleur seront reçus et acquittés par les modules ; il n'y aura donc pas de ré-émission de messages de la part du contrôleur.

Ces suppositions nous permettent de déterminer les capacités au pire cas des threads `fieldbus_writer` et `core_manager` en fonction du nombre de messages lus sur le bus. Ce nombre noté m sera donc compris entre 0 et 6.

Au pire cas, `fieldbus_writer` aura $(3 + m)$ notifications à traiter : m notifications d'envoi d'acquittement, et 3 notifications d'envoi de messages unicast. Par conséquent, sa capacité sera de $(3 + m)$ ms.

Au pire cas, `core_manager` aura $(2 + m)$ notifications à traiter : m notifications provenant de `fieldbus_reader`, une de `task_scheduler`, et une de `gui_controller`. Par conséquent, son temps d'exécution sera de $(4 + 2m)$ ms.

Pour les explications qui vont suivre, nous considérons que la configuration du bus de terrain établit un débit global de 9600 bauds. Le délai de transmission d'un octet dans une telle configuration est de 1 041.7 μs. Toutes les tâches démarreront à $t = 0$, et les échéances seront égales aux périodes d'activation. Enfin, nous n'utiliserons que des tâches périodiques.

La tâche de lecture du bus est la plus critique de notre jeu de tâches. Sa période d'activation est fixée à 21 ms, ce qui correspond au temps de transfert d'un message de taille maximale. La politique d'ordonnancement de cette tâche sera `SCHED_FIFO` avec une priorité maximale de 99. Il est à noter que, dans une telle configuration du bus de terrain, un seul message sera lu par déclenchement.

La tâche d'écriture sur le bus est la seconde tâche la plus critique de notre jeu de tâches. Afin d'être synchrone avec la tâche de lecture du bus, nous fixons sa période d'activation à 21 ms. La politique d'ordonnancement sera aussi `SCHED_FIFO`. Par contre la priorité sera légèrement inférieure à celle de `fieldbus_reader` afin d'assurer la primauté d'exécution. Nous fixons donc la priorité à 90.

`core_manager` fait aussi partie des tâches critiques du système. Nous la considérons secondaire par rapport aux tâches chargées des interactions avec le bus de terrain. Cette fois encore, nous fixons la période à 21 ms afin d'être synchrone avec la tâche de lecture du bus. La politique d'ordonnancement sera `SCHED_FIFO` et la priorité est fixée à 50 afin d'être déclenchée après les tâches de gestion du bus.

Pour le thread qui évalue la présence ou non d'actions planifiées à réaliser, nous fixons la période d'activation sur une minute. En effet, l'interface de configuration ne permettra pas de descendre en dessous de ce niveau de granularité. Nous choisissons la politique d'ordonnancement `SCHED_RR` avec une priorité fixe de 25.

Pour le thread chargé de la gestion de l'interface graphique, nous avons besoin d'un niveau de réactivité similaire à un battement de paupière, c'est à dire environ 150 ms. Nous fixons donc la période d'activation à cette valeur. Enfin, nous choisissons la politique d'ordonnancement `SCHED_RR` avec une priorité fixe de 25. Ainsi, `task_scheduler` et `gui_controller` se partageront à tour de rôle le temps CPU.

Le tableau 4.1 récapitule l'ensemble des propriétés d'ordonnancement des tâches de notre système.

thread	S_i	C_i (ms)	P_i (ms)	D_i (ms)	politique d'ordonnancement	priorité
fieldbus_reader	0	2 ms	21	21	SCHED_FIFO	99
fieldbus_writer	0	4 ms	21	21	SCHED_FIFO	90
core_manager	0	6 ms	21	21	SCHED_FIFO	50
task_scheduler	0	1 ms	60000	60000	SCHED_RR	25
gui_controller	0	10 ms	150	150	SCHED_RR	25

TAB. 4.1 – Jeu de tâches pour les tests d'ordonnancement

4.3 Implantation du modèle du contrôleur

Pour réaliser des tests de faisabilité ainsi que des simulations d'ordonnancement, AADLInspector a besoin de connaître les propriétés relatives à l'ordonnancement. Ces propriétés sont notamment utilisées par le moteur de Cheddar v3.

Dans cette partie, nous présentons l'implantation de notre modèle de contrôleur central pour une configuration du bus avec un débit global de 9600 bauds. Le paramétrage se situe essentiellement sur deux types d'entités : le processeur et les threads. Après avoir présenté l'implantation du processeur et des différents threads, nous décrivons la déclaration de la ressource partagée entre `fieldbus_reader` et `fieldbus_writer`. Enfin, nous détaillons les modifications que nous avons dû appliquer sur notre modèle initial afin de pouvoir réaliser les tests d'ordonnancement.

4.3.1 Implantation du processeur de la carte électronique

Pour réaliser des analyses sur l'ordonnancement d'un jeu de tâches, Cheddar a besoin de connaître le protocole d'ordonnancement du processeur. Pour cela, il faut utiliser la propriété **Scheduling_Protocol**. De base, plusieurs protocoles sont disponibles : **Rate Monotonic**, **Deadline Monotonic**, **Earliest Deadline First**, **Least Laxity First**, **POSIX 1003 Highest Priority First**, et **Pipeline User Defined**. Il est aussi possible de préciser si la préemption est activée au moyen de la propriété **Preemptive_Scheduler**. La préemption caractérise la capacité du processeur à arrêter le traitement d'une tâche pour en exécuter une autre. Enfin, si des tâches sont déclarées avec une politique d'ordonnancement Round-Robin (`SCHED_RR`), il est nécessaire d'indiquer le quantum via la propriété **Scheduler_Quantum**. Le quantum correspond à la durée maximale durant laquelle une tâche peut accaparer le processeur sans être préemptée.

Pour le processeur de la carte électronique, nous avons choisi le protocole d'ordonnancement `POSIX 1003 Highest Priority First`. Ce protocole peut être utilisé avec des threads périodiques ou non. Il permet d'ordonnancer les threads selon une priorité et une politique d'ordonnancement à choisir entre `SCHED_FIFO`, `SCHED_RR`, et `SCHED_OTHERS`. L'extrait de code 4.1 présente la déclaration complète du processeur de la carte électronique.

```
processor board_processor
features
    fieldbus_access: requires bus access domotx_fieldbus;
    cpubus_access  : requires bus access board_cpubus;
properties
    Scheduling_Protocol  => (POSIX_1003_Highest_Priority_First_Protocol);
    Preemptive_Scheduler => True;
    Scheduler_Quantum    => 1;
end board_processor;

processor implementation board_processor.with_memory
subcomponents
    -- association de la mémoire
    ram: memory board_ram;
end board_processor.with_memory;
```

Src. 4.1 – Déclaration du processeur de la carte électronique

4.3.2 Implantation des threads du processus du contrôleur

Pour les threads AADL, les propriétés d'ordonnancement sont à préciser au niveau de leur implantation. La propriété **Dispatch_Protocol** permet de préciser le protocole de déclenchement du thread. Cheddar reconnaît les protocoles suivants : **Periodic**, **Aperiodic**, **Sporadic**, et **Poisson**. La capacité du thread est renseignée avec la propriété **Compute_Execution_Time**. Elle prend comme valeur un intervalle de temps, ce qui permet de modéliser des cas où une tâche est déclenchée mais n'a rien à faire. La période d'activation et l'échéance sont respectivement précisées avec les propriétés **Period** et **Deadline**. La priorité du thread et la politique d'ordonnancement sont respectivement renseignées avec les propriétés **Priority** et **POSIX_Scheduling_Policy**. Enfin dans le cas où un thread partage une ressource avec d'autres threads, il est possible de préciser la durée d'accession à la ressource. Pour cela, on utilise la propriété **Bound_On_Data_Blocking_Time**. Cette propriété ne fait pas partie de langage AADL de base. Elle est définie dans le **property set** `Cheddar_Properties` qui est fourni avec Cheddar.

L'extrait de code 4.2 présente l'implantation du thread de lecture du bus.

```
thread implementation fieldbus_reader.impl_periodic
properties
    Dispatch_Protocol       => Periodic;
    Compute_Execution_Time  => 2 ms .. 2 ms;
    Deadline                => 21 ms;
    Period                  => 21 ms;
    Priority                => 99;
    POSIX_Scheduling_Policy => SCHED_FIFO;
    Cheddar_Properties::Bound_On_Data_Blocking_Time => 0 ms;
end fieldbus_reader.impl_periodic;
```

Src. 4.2 – Implantation du thread `fieldbus_reader`

L'extrait de code 4.3 présente l'implantation du thread d'écriture sur le bus.

```
thread implementation fieldbus_writer.impl_periodic
properties
    Dispatch_Protocol       => Periodic;
    Compute_Execution_Time  => 4 ms .. 4 ms;
    Deadline                => 21 ms;
    Period                  => 21 ms;
    Priority                => 90;
    POSIX_Scheduling_Policy => SCHED_FIFO;
    Cheddar_Properties::Bound_On_Data_Blocking_Time => 0 ms;
end fieldbus_writer.impl_periodic;
```

Src. 4.3 – Implantation du thread `fieldbus_writer`

L'extrait de code 4.4 présente l'implantation du thread des traitements internes.

```
thread implementation core_manager.impl_periodic
properties
    Dispatch_Protocol       => Periodic;
    Compute_Execution_Time  => 6 ms .. 6 ms;
    Deadline                => 21 ms;
    Period                  => 21 ms;
    Priority                => 50;
    POSIX_Scheduling_Policy => SCHED_FIFO;
end core_manager.impl_periodic;
```

Src. 4.4 – Implantation du thread `core_manager`

L'extrait de code 4.5 présente l'implantation du thread de gestion des tâches planifiées.

```
thread implementation task_scheduler.impl_periodic
properties
    Dispatch_Protocol       => Periodic;
    Compute_Execution_Time  => 1 ms .. 1 ms;
    Deadline                => 60000 ms;
    Period                  => 60000 ms;
    Priority                => 25;
    POSIX_Scheduling_Policy => SCHED_RR;
end task_scheduler.impl_periodic;
```

Src. 4.5 – Implantation du thread `task_scheduler`

L'extrait de code 4.6 présente l'implantation du thread de gestion de l'interface graphique.

```
thread implementation gui_controller.impl_periodic
properties
    Dispatch_Protocol       => Periodic;
    Compute_Execution_Time  => 10 ms .. 10 ms;
    Deadline                => 150 ms;
    Period                  => 150 ms;
    Priority                => 25;
    POSIX_Scheduling_Policy => SCHED_RR;
end gui_controller.impl_periodic;
```

Src. 4.6 – Implantation du thread `gui_controller`

4.3.3 Accès exclusif au bus de terrain

Lors de l'étape de modélisation, nous avions précisé que `fieldbus_reader` et `fieldbus_writer` se partageaient une ressource exclusive afin de pouvoir accéder au bus de terrain. Pour cela, nous avons déclaré une entité de type data appelée `domotx_shared`. AADL permet de préciser le protocole d'accès concurrent à une data au moyen du mot-clé **Concurrency_Control_Protocol**. Cheddar reconnaît de base plusieurs protocoles d'accès concurrent : **Priority Inheritance**, **Priority Ceiling**, et **Immediate Priority Ceiling**. Comme `fieldbus_reader` et `fieldbus_writer` ne se partagent qu'une seule ressource, le protocole **Priority Inheritance** peut être utilisé. Ce protocole réduit les cas d'inversion de priorité en faisant en sorte qu'une tâche qui bloque une autre tâche plus prioritaire exécute la section critique avec la priorité de la tâche bloquée. L'extrait de code 4.7 présente l'implantation de la ressource partagée entre `fieldbus_reader` et `fieldbus_writer`.

```
data implementation domotx_shared.fieldbus
properties
     Concurrency_Control_Protocol => Priority_Inheritance;
end domotx_shared.fieldbus;
```

Src. 4.7 – Implantation de la ressource partagée `domotx_shared`

4.3.4 Modification du modèle pour compatibilité avec Cheddar

Il semble que AADLInspector rencontre des problèmes d'interfaçage avec le moteur de Cheddar dans certaines conditions. Dans notre cas, nous n'arrivions pas à lancer des

simulations d'ordonnancement à partir de notre modèle : l'application se figeait indéfiniment.

Après plusieurs tests, il s'est avéré que Cheddar v3 ne fonctionne pas bien dans le cas où un système a comme sous-composant un autre système qui contient processeur. La solution à ce problème a consisté à remonter la déclaration du processeur au niveau de l'entité `domotx_control_system` qui modélise le contrôleur central. L'extrait de code 4.8 présente la déclaration de `domotx_control_system` après modification.

```
-- implémentation du système
system implementation domotx_control_system.impl
subcomponents
    domotx_cpu: processor board_processor.with_memory;
    domotx_appli: process dcs_application.impl;

    domotx_bus  : bus domotx_fieldbus;
connections
    -- connexions au bus de terrain
    B_bus_board  : bus access domotx_bus -> domotx_cpu.fieldbus_access;
properties
    Actual_Processor_Binding => (reference (domotx_cpu)) applies to domotx_appli;
end domotx_control_system.impl;
```

Src. 4.8 – Modification du système `domotx_control_system`

Il est à noter que cette modification n'a aucune incidence sur le résultat des tests de faisabilité et des simulations d'ordonnancement. Le modèle complet du contrôleur central se trouve au niveau de l'annexe B.

4.4 Analyse des résultats

Cette partie présente les résultats des tests de faisabilité et des simulations d'ordonnancement réalisés sur le modèle de contrôleur central. Nous avons utilisé la configuration décrite dans la partie précédente.

Après avoir présenté les résultats des tests de faisabilité, nous affichons le résultat de la simulation d'ordonnancement.

4.4.1 Déroulement des tests de faisabilité

AADL propose deux tests de faisabilité via le moteur de Cheddar v3. Le premier est basé sur le taux d'utilisation du processeur, le second correspond au calcul du pire temps de réponse.

AADLInspector fournit trois résultats théoriques pour le test basé sur le taux d'utilisation du processeur :

- l'hyperpériode qui correspond à la période à partir de laquelle l'ordonnancement se répète ;
- l'utilisation du processeur basé sur la période des tâches qui correspond au pourcentage de temps que le processeur passe à dérouler le jeu de tâches ;
- l'utilisation du processeur basé sur l'échéance des tâches qui correspond globalement à la même chose.

Dans le cadre de notre jeu de tâches, AADL n'est pas capable de déterminer l'hyperpériode. Nous pensons que cela s'explique par l'énorme écart entre les périodes. En

effet, nous avons une période minimale de 21 ms (`fieldbus_reader`, `fieldbus_writer`, et `core_manager`) et une période maximale de 60 000 ms (`task_scheduler`).

Concernant le taux d'utilisation du processeur à partir de la période, Cheddar indique une valeur de 63,81%. Il est possible de vérifier cette valeur à partir de l'équation 2.1. Pour un jeu de 5 tâches à ordonnancer, il faut que la valeur du taux d'utilisation soit inférieure ou égale à $5(2^{\frac{1}{5}} - 1)$ c'est à dire 74,43%. Notre jeu de tâches est donc ordonnançable, et il restera suffisamment de temps CPU pour les tâches internes au système d'exploitation.

Le calcul du pire temps de réponse permet de vérifier que l'ensemble des échéances d'un jeu de tâches seront respectées. Le tableau 4.2 présente les résultats de ce calcul par Cheddar v3 à travers AADLInspector.

thread	résultat
fieldbus_reader	2
fieldbus_writer	6
core_manager	12
task_scheduler	13
gui_controller	34

TAB. 4.2 – Pires temps de réponses des tâches

Ces résultats confirment que notre jeu de tâches est ordonnançable : toutes les échéances seront respectées. On peut aussi vérifier que le temps de réponse des threads `fieldbus_reader`, `fieldbus_writer`, et `core_manager` est optimal et correspond au comportement que nous avions prévu au moment de la modélisation. Il est possible de vérifier les résultats obtenus à partir de l'équation 2.2.

4.4.2 Déroulement des simulations d'ordonnancement

AADLInspector permet de réaliser une simulation d'ordonnancement d'un jeu de tâches. Les résultats de cette simulation sont affichés dans un graphe dont l'axe des abscisses correspond au temps. Ce mode de représentation permet de vérifier qu'un jeu de tâches est ordonnançable si toutes les tâches sont complétées pour un temps équivalent à l'hyperpériode. La figure 4.1 permet de voir une partie de la simulation d'ordonnancement sur une durée de 150 ms. Les codes couleurs sont indiqués sur la figure 4.2.

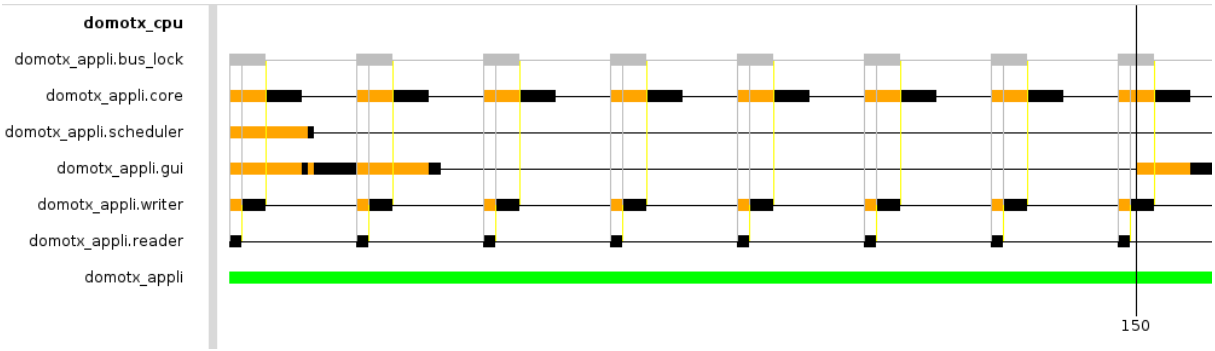

FIG. 4.1 – Extrait de la simulation d'ordonnancement (AADLInspector)

FIG. 4.2 – Codes couleurs utilisés dans la simulation d'ordonnancement

Ce graphe permet de vérifier que `fielbus_reader` est toujours déclenché au début de sa période. Viennent ensuite `fieldbus_writer` et `core_manager`. Enfin, `gui_controller` et `task_scheduler` sont déclenchés en même temps. Ils se partagent alors le temps CPU à chaque expiration du quantum défini dans l'implantation du processeur de la carte graphique.

4.5 Conclusion

Dans ce chapitre, nous avons implanté le jeu de tâches élaboré lors de l'étape de modélisation du contrôleur, puis nous avons réalisé un ensemble de tests et de simulations d'ordonnancement via Cheddar v3. Ces tests ont révélé que, pour la configuration de bus sélectionnée, le système est ordonnançable. Il ne faut cependant pas oublier que l'implantation des threads est basée sur un ensemble de suppositions sur le fonctionnement du contrôleur et sur l'activité du réseau domotique.

Il est clair que dans un cadre de fonctionnement normal, l'activité respective de chaque thread sera différente. Nous avons considéré un cas de fonctionnement aux limites où chaque thread du contrôleur est très sollicité. Pour pousser plus loin notre étude qualificative, il serait intéressant de décrire un comportement dynamique pour chacun des threads déclarés. Ainsi il serait possible de varier la charge de chaque thread et vérifier dans quelles mesures les contraintes temporelles sont respectées.

5 Conclusion générale

L'objectif de ce stage consistait à vérifier en avance de phase de développement s'il était possible d'évaluer le comportement d'un système complexe, à savoir le contrôleur central d'un réseau domotique. Pour cela, un état de l'art a été réalisé sur le domaine des langages de description d'architecture. Cela nous a permis de découvrir le langage AADL avec lequel nous avons proposé une architecture du contrôleur central. Ensuite, à partir d'un ensemble de conjectures sur le fonctionnement du système domotique, nous avons pu utiliser des outils d'analyse afin de réaliser des tests de faisabilité et des simulations d'ordonnancement. Nous avons alors pu vérifier que l'architecture définie dans une configuration donnée permettait de répondre aux exigences non fonctionnelles relatives à l'ordonnancement des différentes tâches, et donc à la réalisation en temps voulu des fonctions du contrôleur.

L'état de l'art des langages de description d'architecture a permis de mieux appréhender la notion d'architecture d'un système. Lorsque l'on considère un système embarqué, il est important de bien distinguer la partie matérielle de la partie logicielle. Le formalisme d'un langage de modélisation est nécessaire pour deux raisons : il permet tout d'abord aux parties prenantes de se comprendre sans ambiguïté, et il rend possible l'utilisation d'outils automatisés. Ces outils permettent de réaliser des analyses sur les systèmes décrits, mais aussi dans le cas de systèmes logiciels de rendre possible la génération de code, ce qui diminue le risque d'erreur au moment de l'implantation. AADL est un langage de plus en plus utilisé dans l'industrie puisqu'il est notamment soutenu par des organismes de taille considérable comme Airbus, ou Boeing.

La prise en compte de besoins fonctionnels et non fonctionnels du contrôleur nous a permis de proposer un modèle d'architecture écrit avec le langage AADL. Nous avons pu partir d'une vue générale du système domotique où nous différencions les principales parties constitutives (modules esclaves, bus de terrain, contrôleur central), pour ensuite descendre en profondeur dans la modélisation du contrôleur central. Celui-ci est divisé en deux parties : la carte électronique et le processus même du contrôleur. Au niveau, du processus, nous avons déterminé 5 threads aux objectifs spécifiques et dont les interactions réalisent les besoins fonctionnels du contrôleur.

Ce modèle établi, nous avons pu réaliser un ensemble de tests afin d'évaluer son comportement temporel. Pour cela, nous avons simplifié légèrement le modèle, et convenu d'un comportement statique au pire cas pour une configuration de bus donnée. Cela nous a permis d'implanter notre modèle et d'utiliser AADLInspector afin de réaliser des tests de

faisabilité et des simulations d'ordonnancement en nous basant sur le moteur de Cheddar v3. Nous avons ainsi pu vérifier que le jeu de tâches défini était bel et bien ordonnançable.

Pour la suite, il sera intéressant de réfléchir à une configuration du jeu de tâches plus proche d'une habitation réelle. Pour cela, il nous faudra obtenir des statistiques d'utilisation des équipements électriques et modifier l'implantation du système afin de vérifier si les contraintes temporelles sont toujours respectées. La méthodologie restera cependant la même. Il sera aussi intéressant de faire varier la taille du réseau considéré afin d'évaluer la charge induite en fonction du nombre de modules esclaves et de la configuration du bus de terrain.

A Modélisation haut-niveau du contrôleur central en AADL

```
package domotx_v1
public

-------------------------------------------------------------------------------
--
-- DÉCLARATION DES DONNÉES
--
-------------------------------------------------------------------------------

-- déclaration des messages échangés sur le bus de terrain
data bus_message
end bus_message;

-- messages émis par les modules à destination du contrôleur
data implementation bus_message.module_signal
end bus_message.module_signal;

-- acquittement d'un signal de module par le contrôleur
data implementation bus_message.module_signal_ack
end bus_message.module_signal_ack;

-- messages émis par le contrôleur à destination d'un ou de plusieurs modules
data implementation bus_message.module_command
end bus_message.module_command;

-- acquittement d'un message de contrôleur par un module (unicast uniquement)
data implementation bus_message.module_command_ack
end bus_message.module_command_ack;

-------------------------------------------------------------------------------
--
-- DÉCLARATION DU BUS DE TERRAIN
--
-------------------------------------------------------------------------------

-- déclaration du bus de terrain
bus domotx_fieldbus
end domotx_fieldbus;

bus implementation domotx_fieldbus.impl
end domotx_fieldbus.impl;

-------------------------------------------------------------------------------
--
-- DÉCLARATION DES GROUPES DE PORTS
--
-------------------------------------------------------------------------------
feature group dcs_socket
features
    -- signal de module
```

```
    sensor_signal       : in  event data port bus_message.module_signal;
    -- acquittement de signal
    sensor_signal_ack   : out event data port bus_message.module_signal_ack;

    -- message de commande
    actuator_command    : out event data port bus_message.module_command;
    -- acquittement de commande
    actuator_command_ack: in  event data port bus_message.module_command_ack;
end dcs_socket;

feature group dcs_plug inverse of dcs_socket
end dcs_plug;

-------------------------------------------------------------------------------
--
-- DÉCLARATION DES MODULES ESCLAVE
--
-------------------------------------------------------------------------------

-- déclaration d'un module
device slave_module
features
    signals: feature group dcs_plug;

    -- accès au bus pour émettre et recevoir les messages
    fieldbus_access: requires bus access domotx_fieldbus;
end slave_module;

-------------------------------------------------------------------------------
--
-- DÉCLARATION DE LA PARTIE MATÉRIELLE DU CONTROLEUR
--
-------------------------------------------------------------------------------

system dcs_board
features
    fieldbus_access: requires bus access domotx_fieldbus;
end dcs_board;

-------------------------------------------------------------------------------
--
-- DÉCLARATION DE LA PARTIE LOGICIELLE DU CONTROLEUR
--
-------------------------------------------------------------------------------

process dcs_appli
features
    signals: feature group dcs_socket;
end dcs_appli;

-------------------------------------------------------------------------------
--
-- DÉCLARATION DU SYSTÈME GLOBAL (MATÉRIEL + LOGICIEL)
--
-------------------------------------------------------------------------------
system domotx_control_system
end domotx_control_system;

-- implémentation du système
system implementation domotx_control_system.impl
subcomponents

    domotx_appli : process dcs_appli;
    domotx_board : system dcs_board;
    domotx_bus   : bus domotx_fieldbus;

    domotx_module: device  slave_module;

connections
    -- connexions au bus de terrain
    B_bus_board  : bus access domotx_bus -> domotx_board.fieldbus_access;
    B_bus_slave  : bus access domotx_bus -> domotx_module.fieldbus_access;
```

```
    -- connexions des ports
    C_appli_slave: feature group domotx_appli.signals <-> domotx_module.signals;
end domotx_control_system.impl;

end domotx_v1;
```

B Modélisation complète du contrôleur central en AADL

```
package domotx_v2
public

with Cheddar_Properties;

-------------------------------------------------------------------------------
--
-- DÉCLARATION DES DONNÉES
--
-------------------------------------------------------------------------------

-- déclaration des messages échangés sur le bus de terrain
data bus_message
end bus_message;

-- messages émis par les modules à destination du contrôleur
data implementation bus_message.module_signal
end bus_message.module_signal;

-- acquittement d'un signal de module par le contrôleur
data implementation bus_message.module_signal_ack
end bus_message.module_signal_ack;

-- messages émis par le contrôleur à destination d'un ou de plusieurs modules
data implementation bus_message.module_command
end bus_message.module_command;

-- acquittement d'un message de contrôleur par un module (unicast uniquement)
data implementation bus_message.module_command_ack
end bus_message.module_command_ack;

-------------------------------------------------------------------------------

-- déclaration des évènements ihm
data gui_event
end gui_event;

-------------------------------------------------------------------------------

-- déclaration des messages échangés entre les tâches du contrôleur
data domotx_message
end domotx_message;

-------------------------------------------------------------------------------

-- déclaration des verrous sur les données
data domotx_shared
end domotx_shared;

data implementation domotx_shared.fieldbus
```

```
properties
     Concurrency_Control_Protocol => Priority_Inheritance;
end domotx_shared.fieldbus;

-------------------------------------------------------------------------------
--
-- DÉCLARATION DU BUS DE TERRAIN
--
-------------------------------------------------------------------------------

-- déclaration du bus de terrain
bus domotx_fieldbus
end domotx_fieldbus;

bus implementation domotx_fieldbus.impl
end domotx_fieldbus.impl;

-------------------------------------------------------------------------------
--
-- DÉCLARATION DES GROUPES DE PORTS
--
-------------------------------------------------------------------------------
feature group dcs_socket
features
    -- signal de module
    sensor_signal       : in  event data port bus_message.module_signal;
    -- acquittement de signal
    sensor_signal_ack   : out event data port bus_message.module_signal_ack;

    -- message de commande
    actuator_command    : out event data port bus_message.module_command;
    -- acquittement de commande
    actuator_command_ack: in  event data port bus_message.module_command_ack;
end dcs_socket;

feature group dcs_plug inverse of dcs_socket
end dcs_plug;

-------------------------------------------------------------------------------
--
-- DÉCLARATION DES MODULES ESCLAVE
--
-------------------------------------------------------------------------------

-- déclaration d'un module
device slave_module
features
    signals: feature group dcs_plug;

    -- accès au bus pour émettre et recevoir les messages
    fieldbus_access: requires bus access domotx_fieldbus;
end slave_module;

device implementation slave_module.impl_periodic
properties
    Dispatch_Protocol => Periodic;
end slave_module.impl_periodic;

device implementation slave_module.impl_aperiodic
properties
    Dispatch_Protocol => Aperiodic;
end slave_module.impl_aperiodic;

-------------------------------------------------------------------------------
--
-- DÉCLARATION DE LA PARTIE MATÉRIELLE DU CONTROLEUR
--
-------------------------------------------------------------------------------

-- déclaration du bus du processeur
bus board_cpubus
end board_cpubus;
```

```
bus implementation board_cpubus.impl
end board_cpubus.impl;

-------------------------------------------------------------------------------

-- déclaration de la mémoire
memory board_ram
end board_ram;

-------------------------------------------------------------------------------

-- déclaration du processeur
processor board_processor
features
    fieldbus_access: requires bus access domotx_fieldbus;
    cpubus_access  : requires bus access board_cpubus;
properties
    Scheduling_Protocol  => (POSIX_1003_Highest_Priority_First_Protocol);
    Preemptive_Scheduler => True;
    Scheduler_Quantum    => 1;
end board_processor;

processor implementation board_processor.with_memory
subcomponents
    ram: memory board_ram;
end board_processor.with_memory;

-------------------------------------------------------------------------------

-- déclaration de la carte
system dcs_board
features
    fieldbus_access: requires bus access domotx_fieldbus;
end dcs_board;

system implementation dcs_board.impl
subcomponents
    board_cpu: processor board_processor.with_memory;
    board_bus: bus board_cpubus.impl;
connections
    B1: bus access fieldbus_access -> board_cpu.fieldbus_access;
    B2: bus access board_bus -> board_cpu.cpubus_access;
end dcs_board.impl;

-------------------------------------------------------------------------------
--
-- DÉCLARATION DE LA PARTIE LOGICIELLE DU CONTROLEUR
--
-------------------------------------------------------------------------------

-------------------------------------------------------------------------------
-- DÉCLARATION DES TÂCHES
-------------------------------------------------------------------------------

-- déclaration du thread chargé de lire sur le bus de terrain les messages
-- émis par les modules, et de retourner un acquittement pour les signaux
-- corrects
thread fieldbus_reader
features
    -- signal de module
    signal             : in  event data port bus_message.module_signal;
    -- acquittement de commande
    command_ack        : in  event data port bus_message.module_command_ack;

    -- émission de notifications vers les autres managers
    notification_output: out event data port domotx_message;

    -- accès exclusif sur le bus de terrain
    fieldbus_lock      : requires data access domotx_shared.fieldbus;
end fieldbus_reader;

thread implementation fieldbus_reader.impl_periodic
properties
```

```
    Dispatch_Protocol       => Periodic;
    Compute_Execution_Time  => 0 ms .. 2 ms;
    Deadline                => 21 ms;
    Period                  => 21 ms;
    Priority                => 99;
    POSIX_Scheduling_Policy => SCHED_FIFO;
    Cheddar_Properties::Bound_On_Data_Blocking_Time => 0 ms;
end fieldbus_reader.impl_periodic;

-------------------------------------------------------------------------------

-- déclaration du thread chargé d'envoyer des commandes à destination d'un
-- ou de plusieurs modules
thread fieldbus_writer
features
    -- commande pour un ou plusieurs modules
    command             : out event data port bus_message.module_command;
    -- acquittement de signal
    signal_ack          : out event data port bus_message.module_signal_ack;

    -- réception des notifications émises par d'autres managers
    notification_input  : in  event data port domotx_message;

    -- accès exclusif sur le bus de terrain
    fieldbus_lock       : requires data access domotx_shared.fieldbus;
end fieldbus_writer;

thread implementation fieldbus_writer.impl_periodic
properties
    Dispatch_Protocol       => Periodic;
    Compute_Execution_Time  => 0 ms .. 2 ms;
    Deadline                => 21 ms;
    Period                  => 21 ms;
    Priority                => 90;
    POSIX_Scheduling_Policy => SCHED_FIFO;
    Cheddar_Properties::Bound_On_Data_Blocking_Time => 0 ms;
end fieldbus_writer.impl_periodic;

-------------------------------------------------------------------------------

-- déclaration du thread chargé des interactions avec l'interface graphique
thread gui_controller
features
    -- prise en compte évènement ihm
    event_input         : in  event data port gui_event;
    -- mise à jour de l'affichage
    refresh_output      : out event data port domotx_message;

    -- réception des notifications émises par d'autres managers
    notification_input : in  event data port domotx_message;
    -- émission de notifications vers les autres managers
    notification_output: out event data port domotx_message;
end gui_controller;

thread implementation gui_controller.impl_periodic
properties
    Dispatch_Protocol       => Periodic;
    Compute_Execution_Time  => 10 ms .. 10 ms;
    Deadline                => 150 ms;
    Period                  => 150 ms;
    Priority                => 25;
    POSIX_Scheduling_Policy => SCHED_RR;
end gui_controller.impl_periodic;

-------------------------------------------------------------------------------

-- déclaration du thread chargé d'envoyer des notifications de tâches à réaliser
-- lorsque le moment est venu
thread task_scheduler
features
    -- message à traiter par le core manager
    notification_output: out event data port domotx_message;
end task_scheduler;
```

```
thread implementation task_scheduler.impl_periodic
properties
    Dispatch_Protocol       => Periodic;
    Compute_Execution_Time  => 0 ms .. 2 ms;
    Deadline                => 60000 ms;
    Period                  => 60000 ms;
    Priority                => 25;
    POSIX_Scheduling_Policy => SCHED_RR;
end task_scheduler.impl_periodic;

-------------------------------------------------------------------------------

-- déclaration du thread principal chargé de traiter les notifications émises
-- par les autres tâches
thread core_manager
features
    -- réception des notifications émises par d'autres managers
    notification_input : in  event data port domotx_message;
    -- émission de notifications vers les autres managers
    notification_output: out event data port domotx_message;
end core_manager;

thread implementation core_manager.impl_periodic
properties
    Dispatch_Protocol       => Periodic;
    Compute_Execution_Time  => 0 ms .. 4 ms;
    Deadline                => 21 ms;
    Period                  => 21 ms;
    Priority                => 50;
    POSIX_Scheduling_Policy => SCHED_FIFO;
end core_manager.impl_periodic;

-------------------------------------------------------------------------------
-- DÉCLARATION DU PROCESSUS
-------------------------------------------------------------------------------

process dcs_application
features
    signals: feature group dcs_socket;
end dcs_application;

process implementation dcs_application.impl
subcomponents
    reader   : thread fieldbus_reader.impl_periodic;
    writer   : thread fieldbus_writer.impl_periodic;
    gui      : thread gui_controller.impl_periodic;
    scheduler: thread task_scheduler.impl_periodic;
    core     : thread core_manager.impl_periodic;

    bus_lock : data domotx_shared.fieldbus;

connections
    -- connexions avec l'extérieur
    C_sensor_reader  : port signals.sensor_signal -> reader.signal;
    C_actuator_reader: port signals.actuator_command_ack -> reader.command_ack;
    C_writer_sensor  : port writer.signal_ack -> signals.sensor_signal_ack;
    C_writer_actuator: port writer.command -> signals.actuator_command;

    -- connexions internes au contrôleur
    C_reader_core   : port reader.notification_output -> core.notification_input;
    C_reader_writer : port reader.notification_output -> writer.notification_input;
    C_gui_core      : port gui.notification_output  -> core.notification_input;
    C_scheduler_core: port scheduler.notification_output -> core.notification_input;
    C_core_gui      : port core.notification_output -> gui.notification_input;
    C_core_writer   : port core.notification_output -> writer.notification_input;

    -- accès exclusif au bus de terrain
    D_fieldbus_1: data access bus_lock -> reader.fieldbus_lock;
    D_fieldbus_2: data access bus_lock -> writer.fieldbus_lock;

end dcs_application.impl;
```

```
-------------------------------------------------------------------------------
--
-- DÉCLARATION DU SYSTÈME GLOBAL (MATÉRIEL + LOGICIEL)
--
-------------------------------------------------------------------------------
system domotx_control_system
end domotx_control_system;

-- implémentation du système
system implementation domotx_control_system.impl
subcomponents

    domotx_appli: process dcs_application.impl;
    domotx_board: system dcs_board.impl;

    domotx_bus  : bus domotx_fieldbus;

connections
    -- connexions au bus de terrain
    B_bus_board  : bus access domotx_bus -> domotx_board.fieldbus_access;

properties
    Actual_Processor_Binding => (reference (domotx_board.board_cpu)) applies to
        domotx_appli;

end domotx_control_system.impl;

end domotx_v2;
```

Références bibliographiques

[42010, 2011] 42010, I. (2011). *ISO/IEC/IEEE 42010*. `http://www.iso-architecture.org/42010`. Accessed : 2014-05-10.

[Allen, 1997] Allen, R. (1997). *A Formal Approach to Software Architecture*. Thèse de Doctorat, Carnegie Mellon, School of Computer Science. Issued as CMU Technical Report CMU-CS-97-144.

[Audsley et al., 1993] Audsley, N., Burns, A., Richardson, M., Tindell, K., and Wellings, A. J. (1993). *Applying New Scheduling Theory to Static Priority Pre-Emptive Scheduling*. Software Engineering Journal, volume 8, pages 284–292.

[Bozzano et al., 2010] Bozzano, M., Cimatti, A., Katoen, J.-P., Nguyen, V. Y., Noll, T., Roveri, M., and Wimmer, R. (2010). *A Model Checker for AADL*. CAV, pages 562–565.

[Chkouri, 2010] Chkouri, M. Y. (2010). *Modélisation des systèmes temps-réel embarqués en utilisant AADL pour la génération automatique d'applications formellement vérifiées*. Thèse de Doctorat, Université de Grenoble.

[Dissaux and Singhoff, 2008] Dissaux, P. and Singhoff, F. (2008). *Stood and Cheddar : AADL as a pivot language for analysing performances of real time architectures*. Proceedings of the European Real Time System conference. Toulouse, France.

[Feiler et al., 2006] Feiler, P. H., Gluch, D. P., and Hudak, J. J. (2006). *The Architecture Analysis & Design Language (AADL) : An Introduction*. Rapport technique, Software Engineering Institute.

[Group, 2000] Group, I. A. W. (2000). *IEEE Std 1471-2000, Recommended practice for architectural description of software-intensive systems*. Rapport technique, IEEE.

[Hudar and Feiler, 2007] Hudar, J. J. and Feiler, P. (2007). *Developping AADL Models for Control Systems : A Practitioner's Guide*. Rapport technique, Software Engineering Institute.

[Hugues and Singhoff, 2009] Hugues, J. and Singhoff, F. (2009). *Développement de systèmes à l'aide d'AADL - Ocarina/Cheddar*. L'école d'été temps réel.

[Liu and Layland, 1973] Liu, C. L. and Layland, J. W. (1973). *Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment*. J. ACM, volume 20, numéro 1, pages 46–61.

[Malavolta et al., 2013] Malavolta, I., Lago, P., Muccini, H., Pelliccione, P., and Tang, A. (2013). *What Industry Needs from Architectural Languages : A Survey*. IEEE Trans. Software Eng., volume 39, numéro 6, pages 869–891.

[Medvidovic, 1997] Medvidovic, N. (1997). *A Classification and Comparison Framework for Software Architecture Description Languages*. IEEE TRANSACTIONS ON SOFTWARE ENGINEERING, volume 26, pages 70–93.

[Pouiller, 2011] Pouiller, J. (2011). *Cours temps-réel*. Rapport technique, Sysmic.

[Singhoff, 2014] Singhoff, F. (2014). *Real-Time Scheduling analysis*. `http://beru.univ-brest.fr/~singhoff/ENS/USTH/rt.html`. Accessed : 2014-08-05.

[Singhoff et al., 2004] Singhoff, F., Legrand, J., Nana, L., and Marcé, L. (2004). *Cheddar : A Flexible Real Time Scheduling Framework*. Ada Lett., volume XXIV, numéro 4, pages 1–8.

[Stankovic, 1988] Stankovic, J. (1988). *Misconceptions about real-time computing*. IEEE Computer, volume 21, pages 10–19.

[Tréflévénet, 2014] Tréflévénet (2014). *TréflévéNet*. `http://www.treflevenet.fr`. Accessed : 2014-05-08.