2ème Année de Master

Logiciels pour les Systèmes Embarqués

Rapport de stage

Approche hybride pour l'optimisation de simulations d'ordonnancement de systèmes temps-réels

Présenté par :

Yann Allain

Travail réalisé en 2017 sous la direction
de Pierre Dissaux et Frank Singhoff

2ème Année de Master

Logiciels pour les Systèmes Embarqués

Rapport de stage

Approche hybride pour l'optimisation de simulations d'ordonnancement de systèmes temps-réels

Présenté par :

Yann Allain

Sous la supervision de :

M. Frank Singhoff	Professeur à l'Université de Brest Occidentale	Directeur de stage
M. Pierre Dissaux	Directeur d'Ellidiss Technologies	Tuteur de stage

Mars - Juillet 2017

1 Remerciements

Je tiens à remercier tout particulièrement mon professeur et directeur de stage M. Frank Singhoff pour ses conseils ainsi que M. Pierre Dissaux pour m'avoir accueilli au sein de son entreprise et pour ses conseils avisés. Je remercie également M. Arnaud Schach, M. Jérome Legrand, M. Olivier Marc, M. Ahcène Bounceur et M. Stéphane Rubini pour les nombreux conseils qu'il ont pu m'apporter lors de ce stage et qui m'ont permis de cibler mes recherches et de mener ce travail à bien dans les meilleures conditions possibles.

2 Introduction

Ce document constitue le rapport de mon stage de fin d'études effectué à Brest au sein de l'entreprise Ellidiss Technologies, spécialisée dans la conception d'outils pour le développement des systèmes critiques à prédominance logicielle, sous la direction conjointe de Pierre Dissaux (Directeur et fondateur de l'entreprise) et Franck Singhoff (Professeur, chargé de recherche à l'UBO / Lab-STICC).

Un système temps réel est soumis comme son nom l'indique, à des contraintes temporelles. Ces contraintes temporelles sont principalement liées à l'exactitude attendue de l'exécution du système en terme de respect de ces contraintes. En effet, nous attendons d'un système temps réel qu'il respecte les contraintes temporelles lui-étant fixées. C'est donc là, la valeur réelle d'un système temps réel. De nombreux systèmes temps réels sont dit critiques. Cela s'explique par la criticité du travail qu'ils effectuent et qui se traduit par un risque de mise en danger des personnes, de l'environnement ou de matériels si les contraintes temporelles du système ne sont pas respectées. Si nous parlons de système de freinage au sein d'un véhicule nous pouvons instantanément comprendre qu'il n'est pas acceptable d'avoir un freinage qui s'effectue plusieurs secondes après que le conducteur ait déclenché la pédale de frein. En effet, nous attendons une réponse du système de freinage qui est immédiate, sans quoi le système devient caduc et dans ce cas présent met en danger le conducteur. Un système temps réel, qu'il soit critique ou non possède donc des contraintes temporelles qui peuvent être plus ou moins rigides et dont le rôle de l'ordonnanceur utilisé au sein du système est d'essayer de garantir le respect de ces contraintes. C'est en effet ce composant fondamental du système qui pour chaque processeur, va en définir l'ordonnancement ; ce qui revient à définir de quelle manière agencer les exécutions des tâches sur chaque processeur. Il existe des moyens d'analyser le modèle d'ordonnancement d'un système donné et d'établir l'ordonnançabilité de celui-ci, c'est-à-dire que les contraintes temporelles et autres critères fixés sont respectés et donc que l'ordonnancement est faisable. Dans ce but, nous pouvons appliquer différentes méthodes d'analyse tel que les tests de faisabilités lorsque ceux-ci sont applicables au modèle considéré et ainsi déterminer l'ordonnançabilité du système. Par ailleurs, il est possible de caractériser le comportement d'un système et de l'évaluer par simulation au sein d'un simulateur d'ordonnancement.

Il est possible de modéliser la simulation de différentes manières. Ainsi pour une simulation dite *time-driven* nous avons une variable modélisant le temps courant, incrémentée à étapes fixes. À chaque incrémentation on vérifie quels sont les évènements pour l'unité de temps courante et ceux-ci sont générés. Ces actions de vérifications sont effectuées à chaque unité de temps. De manière opposée, une simulation *event-driven* effectue les calculs pour la génération d'évènement seulement lorsqu'un évènement se produit. Ainsi, aucun calcul lié à la vérification de la présence ou non d'évènement n'est effectué à chaque unité de temps. La simulation effectue ainsi un «saut» à la prochaine unité de temps où un évènement se produit.

Ce stage porte principalement sur l'étude des simulations d'ordonnancement time-driven, qui en fonction des systèmes considérés peuvent posséder un pourcentage important de la simulation sur des unités de temps qui sont creuses en terme d'évènements, et à la manière d'éviter ces *temps creux* au sein des calculs de la simulation. En effet, comme nous le verrons au début de ce document, certains simulateurs d'ordonnancement ont déjà abordé la question avec des résultats plus ou moins probants en terme de précision, ainsi que dans la limitation des algorithmes d'ordonnancement supportés. Nous proposerons une solution au travers de ce rapport permettant d'éviter les calculs sur ces unités de temps creux mais aussi d'étendre ce fonctionnement à d'autres motifs de l'ordonnancement. Ainsi, la conception d'une simulation d'ordonnancement hybride mêlant les avantages de la simulation time-driven aux avantages de la simulation event-driven et permettant donc de garder précision de l'ordonnancement et amplitude du nombre d'algorithmes d'ordonnancement supportés n'a été que très peu étudié. Nous verrons l'application de cette solution au travers d'un cas d'étude sur lequel s'est basé le développement initial de la solution puis sur un cas pratique [1] de mise en œuvre de la solution sur un autre simulateur d'ordonnancement au fonctionnement radicalement différent.

1. Le cas pratique permettra de valider la solution adoptée sur un autre type de simulateur.

Table des matières

3 Contexte

La société Ellidiss Technologies propose différents outils logiciels pour l'aide a la conception de logiciels critiques qu'elle produit, tel que le logiciel STOOD supportant les standards *Hierarchical Object Oriented Design (HOOD)* et *Architecture Analysis and Design Language (AADL)*[2], permettant de modéliser un système critique, composants matériels et logiciels, de façon graphique ou textuelle et de générer le code dans différents langages. La majorité de son domaine d'activité tourne autour de la modélisation de systèmes critiques via le langage AADL.

Les intérêts de l'entreprise Ellidiss Technologies et de l'UBO se recoupent dans la mesure où les deux acteurs souhaitent optimiser le temps de la simulation et diminuer le nombre d'évènements, que ce soit une simulation statique (calcul de l'ensemble des données puis affichage) mais surtout dynamique (calcul d'une unité de temps puis affichage et ainsi de suite). En effet, comme nous le verrons dans la partie suivante, en fonction du taux d'utilisation du ou des processeurs du système, une partie inversement proportionnelle du temps est passée sur des périodes d'inactivité. De plus, nous verrons par la suite que même la génération d'évènements de certaines « activités » du système peuvent êtres évitées.

Trois acteurs principaux partagent des intérêts dans ce sujet de stage. Le premier acteur est le laboratoire STICC (CNRS) de l'UBO qui cherche à optimiser le temps sur les calculs de la simulation et du nombre d'évènements mais aussi d'ouvrir la voie à une simulation multi-échelle. Ces différents objectifs seront explicités et approfondis plus après. Le second acteur, Ellidiss Technologies souhaite aussi améliorer les temps de simulation en statique et dynamique ainsi que le nombre d'évènements générés et harmoniser le fonctionnement de deux logiciels de simulation différents Cheddar[3] et Marzhin qu'elle utilise au sein de son logiciel AADLInspector. Le dernier acteur est l'entreprise Virtualys dirigée par Olivier Marc et qui fourni le logiciel de simulation Marzhin à son partenaire Ellidiss Technologies.

2. [1] P. Feiler, B. Lewis and S. Vestal, *The SAE AADL standard : A basis for model-based architecture driven embedded systems engineering*, Workshop on Model-Driven Embedded Systems, 2003.

3. [2] F. Singhoff, J. Legrand, L. Nana, and L. Marcé, *Cheddar : A Flexible Real Time Scheduling Framework*, International ACM SIGADA Conference, Atlanta, November 2004.

3.1 But et Objectifs

Il convient de présenter les différentes problématiques qui balisent l'ensemble du travail effectué. Cependant, il faut en tant que lecteur savoir que les objectifs énoncés ci-dessous ont été introduits au fur et à mesure de l'avancement du stage et en fonction des discussions que les différents acteurs de ce stage ont pu avoir.

3.1.1 Elimination des temps creux

La première problématique que nous allons définir ici est le socle principal du stage. En effet, comme nous l'avons vu dans l'introduction du contexte du stage, c'est le point sur lequel se recoupent les intérêts des intervenants de l'UBO, d'Ellidiss Technologies mais aussi de Virtualys pour le cas pratique que nous verrons dans la partie qui lui est dédiée. Cette problématique vise, comme nous l'avons déjà succinctement abordé, à éliminer les temps creux, c'est-à-dire les temps ou aucune tâche n'est en cours d'exécution sur un processeur donné, et donc que durant cette unité de temps, le processeur est inactif et aucun évènement d'ordonnancement n'est généré. Nous chercherons donc une manière de modifier le fonctionnement d'un simulateur d'ordonnancement time-driven, dans le cas d'étude qui nous intéresse ici, Cheddar – que nous décrirons plus après –, afin de garantir une diminution du nombre de temps creux et donc d'obtenir une exécution de la simulation plus rapide et de limiter la génération d'évènements inutiles en terme de compréhension et d'analyse de l'ordonnancement généré. Nous chercherons aussi à savoir si le schéma d'optimisation que nous auront défini pour les temps creux est applicable à d'autres motifs de l'ordonnancement, et si optimisation possible, de quelle manière l'implémenter. De plus, des tests seront effectués afin de vérifier que la solution apporté est valable sur différentes architectures et avec différents algorithmes d'ordonnancement ; la polyvalence de la solution étant un des objectifs majeurs du stage. Nous chercherons pour terminer à implémenter la solution de notre cas d'étude au sein d'un cas pratique, via l'implémentation de la solution au sein du logiciel Marzhin de l'entreprise Virtualys, partenaire d'Ellidiss Technologies, possédant un fonctionnement différent du logicel Cheddar et ainsi confirmer ou non la simplicité d'implémentation de la solution au sein d'autres simulateurs d'ordonnancement.

3.1.2 Problématique d'intégration

Cette seconde problématique est liée à des besoins internes de l'entreprise Ellidiss Technologies et peut-être décrite comme étant le besoin d'avoir une certaine homogénéité entre deux logiciels utilisés : Cheddar et Marzhin. En effet, ces deux logiciels que nous aborderons plus tard, permettent d'effectuer de la simulation d'ordonnancement mais chacun avec une stratégie différente. Ainsi, les résultats peuvent différer pour un même modèle entre les deux logiciels. Ainsi, Ellidiss Technologies utilise les deux logiciels au sein de son outil AADLInspector [4] afin d'avoir un comparatif et donc une meilleure analyse et compréhension du modèle étudié. L'outil Cheddar calcule l'ensemble de la table d'évènement [5] puis cette table est affichée au sein d'AADLInspector. Marzhin quand à lui effectue une simulation pas à pas ou chaque pas de la simulation est une nouvelle unité de temps. Nous chercherons premièrement à transformer la simulation statique de Cheddar en simulation dynamique semblable à Marzhin, ce qui permettra d'avoir une comparaison en temps réel des deux ordonnancements ou tout simplement d'inter-changer le logiciel utilisé au sein d'AADLInspector si un seul chronogramme est affiché. De plus, avec une simulation dynamique nous pourrons mieux percevoir les amélioration apportées par l'élimination des temps creux. En conclusion nous chercherons ici à répondre à un besoin interne de l'entreprise mais qui de plus permettra une meilleure visualisation des résultats de l'ordonnancement de notre cas d'étude après implémentation de la solution.

3.1.3 Simulation multi-échelle

La dernière problématique est un objectif secondaire qui a été abordé dans les échanges que nous avons pu avoir avec les différents intervenants et concerne la translation d'une simulation d'ordonnancement à une seule dimension temporelle vers une simulation multi-échelle. Cet objectif n'ayant que très peu été abordé et son intérêt par rapport à l'objectif principal étant plus faible, il n'a pas profité de la charge de travail qu'a pu recevoir l'objectif principal basé sur l'optimisation des temps creux. Cependant, nous chercherons à savoir si l'élimination des temps creux apporte ou non un avantage pour la simulation multi-échelle ainsi que de chercher une méthode permettant de représenter de grands entiers pour notre cas d'étude

4. AADLInspector est un logiciel édité par Ellidiss Technologies et offrant entre autre, la possibilité d'éditer des projets AADL, leur vérification syntaxique ainsi qu'analyse et simulation d'ordonnancement.

5. La table d'évènement peut-être représentée au format *Extensible Markup Language (XML)* avec pour chaque unité de temps les différents types d'évènements et informations lui-étant lié.

Cheddar, programmé en *Ada*.

3.2 Portée

Le but de ce travail, effectué dans le cadre de ce stage, est de produire une solution répondant aux différents objectifs énoncés auparavant. Les contraintes de temps du stage, inhérentes à la durée du stage en lui-même dictent la charge de travail pouvant être déployée durant le stage. De part cette contrainte, il n'est pas faisable d'offrir le même niveau de complétude et de fonctionnalités que ce qui aurait être effectué avec une durée plus importante. Ainsi, la vérification des performances des optimisations mises en place, et qui répondent à la problématique principale, n'ont pas pu être effectuées. Une analyse de performance supplémentaire de la solution proposée en réponse à cette problématique, que ce soit en terme de performance temporelle ou de performance mémoire, serait la bienvenue pour compléter ce travail.

3.3 Méthodologie

Le travail sur le stage a été divisé en trois phases, recherche de documentation sur le sujet de la simulation event-driven appliquée à la théorie de l'ordonnancement et analyse du code des logiciels sur lesquels intervenir, recherche théorique d'une solution et algorithmes utiles à celle-ci, implémentation de la solution sur le cas d'étude et le cas pratique plus tests. Un résumé de l'approche prise dans chacune de ces phases est décrit ci-dessous.

L'implémentation d'une solution pour l'optimisation des temps creux en tendant vers une simulation event-driven requiert une compréhension approfondie du sujet, que ce soit sur la théorie elle-même ou sur les manières d'implémenter ce type de simulation. Plutôt que de directement démarrer le projet sans repère autre que la théorie, nous avons convenu qu'il serait plus judicieux d'étudier des solutions déjà existantes et pouvant apporter des informations bénéfiques pour l'approche finale que nous choisirons. L'architecture, le code source et publications, lorsque disponibles, de différents projets portant sur le domaine de la simulation d'ordonnancement ont été étudiés afin d'avoir une idée sur la meilleure façon de répondre à la problématique principale de ce stage. De plus, une connaissance approfondie du langage Ada utilisé pour notre cas d'étude, Cheddar, était bien-évidemment essentielle au développement. En plus de la connaissance du langage Ada qui a été approfondie, une compréhension du fonctionnement de Cheddar, Marhzin et AADLInspector, et donc de leur code source a été nécessaire afin de connaître les zones sur lesquelles intervenir ainsi que les implications potentielles des futures modifications sur le reste du fonctionnement du logiciel impliqué. Après avoir étudié la simulation event-driven et divers travaux effectués en rapport avec ce que nous cherchons à construire, ainsi

qu'après l'analyse des logiciels sur lesquels des modifications sont à apporter vis à vis des différentes problématiques que nous avons pu définir dans la partie dédiée, une solution théorique a été définie.

Après avoir étudié les différents supports, une approche théorique a été construite. Cette approche a été construire en se basant sur les exigences que nous avons pu définir entre les différents intervenants du stage en rapport avec notre problématique principale et avons donc proposé une solution théorique basée sur un système d'heuristiques permettant d'éviter les temps creux et donc de les « sauter » au sein de la simulation tout en ayant le reste de la simulation sur un fonctionnement time-driven.

Après avoir construit notre approche théorique nous avons implémenté la solution au sein de notre cas d'étude Cheddar, en langage Ada, tout en respectant les exigences étant liée à l'implémentation de la solution au sein de ce logiciel. Des modifications supplémentaires ont été apportées en plus de la problématique principale afin de permettre à la simulation Cheddar de passer en simulation pas à pas de la même manière que le simulateur d'ordonnancement Marzhin. Ainsi, en ayant ajouté un protocole de communication basé sur les sockets nous avons pu faire communiquer Cheddar et AADLInspector afin de piloter la simulation Cheddar et ainsi percevoir de manière graphique les optimisation apportées par notre solution. Une fois le cas d'étude validé par les tests, la solution a été intégrée au cas pratique avec succès au sein du simulateur Marzhin programmé en langage Java et dont le fonctionnement interne diffère de Cheddar. Enfin des tests de validation du fonctionnement de la simulation ont aussi été appliqués à Marzhin.

3.4 Conventions typographiques

Ci-dessous une liste des conventions typographiques utilisées tout au long de ce document.

- Les termes définis dans le glossaire seront écrit avec une police italique la première fois qu'ils seront utilisés, e.g. *terme*.

- Les noms de fichiers seront affichés avec une police monospace,
  e.g. `fichier.suffixe` tout comme les noms de packages et méthodes e.g `Ma_Méthode(..)`

- Les commandes sensées être entrées sur une ligne de commande seront affichées avec une police monospace avec le symbole plus-grand-que en tant que préfixe, e.g `> date`.

- Les extraits de code source seront affichés dans un police monospace avec une coloration syntaxique, voir listing 1.

```
type Heuristic is tagged
        record
                name : Unbounded_String;
        end record;
```

Listing 1: Exemple de code

4 Documentation

Un des points majeurs avant la réalisation d'un projet, quel qu'il soit, est la recherche de travaux précédents. Nous avons la chance en France d'avoir divers organismes de recherche, des financement de la part de l'Union européenne et de la part du Ministère de l'enseignement. Ceux-ci garantissent les travaux de chercheurs dans de multiples domaines. Internet permet quand à lui de rendre disponible les papiers de recherche de différents travaux réalisés à travers le monde sur des plateformes tel qu'arXiv [6] ou autres et qui permettent d'accéder à des papiers spécifiques sans passer par des revues spécialisées. J'ai donc commencé par rechercher des travaux en rapport avec la problématique que nous avons définie. Ainsi, plusieurs projets et articles ont été retenus pour étude. De plus, concepts et méthodologies d'applications concernant la simulation event-driven ont été étudiés [7].

En plus des articles et sites des projets, lorsque ceux-ci semblaient répondre en tout ou partie à notre problématique mais aussi lorsque des éléments étaient manquants, une analyse du code source des projets concernés (lorsque disponible) a été effectuée. Nous recherchons donc des projets ayant implémentés ou proposant une solution à un simulateur d'ordonnancement event-driven. L'objectif principal étant comme nous l'avons vu, de gagner du temps sur la simulation et de limiter le nombre d'évènements générés en évitant les temps creux. De plus, nous cherchons – pour les simulateurs implémentés et où le code source est disponible – si ils prennent en charge de très grands entiers et comment. Ce qui comme nous l'avons vu dans l'annonce des problématiques, permettrait de travailler avec des échelles de temps de grand écart [8]. Un résumé des différents travaux existants selectionnés pour étude est décrit ci-dessous.

6. arXiv est une archive de prépublications électroniques d'articles scientifiques dans différents domaines scientifiques et accessible gratuitement sur internet.

7. [3] Thomas J. Schriber and Daniel T. Brunner, *How Discrete-Event Simulation Software Works*, *Handbook of Simulation*, Edited by Jerry Banks, John Wiley & Sons, N.Y. 1998, p. 765-811.

8. e.g unités de temps liée aux tâches en μs et en s

Voici un résumé sur quatre simulateurs event-driven, nous éviterons d'entrer dans les détails des architectures des différentes solutions et simplement fournir les informations que l'on recherche ici.

4.1 HAPI

Concernant le simulateur HAPI, aucun code source n'a été accessible hors des extraits présentés au sein du papier de recherche. Ainsi, la documentation du papier de recherche [9] nous permet de déterminer que ce projet de simulateur event-driven destiné à l'étude des systèmes temps réels est programmé en *SystemC*[10] et cela permet donc de profiter du *Transaction-Level Modeling (TLM)* permettant de définir un système avec un haut niveau d'abstraction. L'implémentation actuelle de ce simulateur tel que proposé au sein de l'article supporte les algorithmes d'ordonnancement suivant : Fixed-priority Scheduling, Time-Division Scheduling, Time-Division Multiplexing Scheduling, Cooperative Round-Robin. Le nombre d'algorithmes supportés est donc assez limité. Afin de définir un système, les différents composants sont mappés sur un *Data Flow Graph (DFG)* via une API C++. Il semblerait qu'aucun calcul spécifique ne soit effectué afin de déterminer la date de fin d'exécution des tâches ou la date de début de leur exécution. En effet, les évènements sont générés via le noyau SystemC et son fonctionnement interne.

9. [4] Philip Sebastian Kurtin, J.P.H.M. Hausmans, Marco Jan Gerrit Bekooij, *HAPI : An event-driven simulator for real-time multiprocessor systems*, 2016 ACM International Workshop on Software and Compilers for Embedded Systems (SCOPES), New York, 2016, p. 60-66.

10. La librairie SystemC propose une simulation event-driven au travers de son noyau qui traite les différents évènements dans l'ordre généré par les composants de ce même système.

4.2 JsimMAST

Un article est proposé décrivant le projet et celui-ci est open-source. Ainsi, ce simulateur programmé en Java vise à fournir une trace du comportement temporel d'un système temps réel. JsimMAST prend en entrée un modèle d'architecture *MAST*[11] 2.0. et définit avant tout un modèle permettant de décrire un système temps réel distribué ainsi que les contraintes temps réelles inhérentes. Ce modèle event-driven est ce qui lui permet d'établir des dépendances entre les différentes tâches. Ainsi, l'utilisation principale de MAST est de fournir un modèle pour l'analyse d'ordonnançabilité basé sur le pire temps de réponse, le calcul de temps de blocage ou encore l'analyse de sensibilité. Le type manipulé par les dates permet de manipuler de grands entiers.

11. [5] Michael González Harbour, J. Javier Gutiérrez José M. Drake, Patricia López Martínez, .J. Carlos Palencia, *Modeling distributed real-time systems with MAST 2*, Journal of Systems Architecture Volume 59, Issue 6, June 2013 (2012), p. 331-340.

4.3 SimSo

Code source, article [12] et documentation sont disponibles concernant ce simulateur. Le coeur de ce simulateur est basé sur SimPy2, un framework Python de simulation à évènement discret basé sur les process. Cela permet de jouer sur des durées très courtes ou très longues pour le même prix en terme de complexité ce qui est un des points que l'on recherche, comme nous l'avons vu au sein de la définition de la problématique concernant la simulation multi-échelle. Le temps d'exécution d'un job durant la simulation est basé sur un *Execution Time Model (ETM)* [13]. De base, celui-ci est basé sur le *Worst Case Execution Time (WCET)* de la tâche mais différents ETM sont possibles. Aucun mécanisme correspondant à notre objectif ne semble être mis en place au sein du simulateur. Concernant la prise en compte des grands entiers, le programme étant développé en Python, le typage est dynamique et la taille est limitée par la mémoire disponible sur le support sur lesquel la simulation est exécutée.

12. [6] Maxime Chéramy, Pierre-Emmanuel Hladik, Anne-Marie Déplanche, *SimSo : A Simulation Tool to Evaluate Real-Time Multiprocessor Scheduling Algorithms*, 5th International Workshop on Analysis Tools and Methodologies for Embedded and Real-time Systems (WATERS), Jul 2014, Madrid, Spain, 6 p., 2014.

13. [7] Maxime Chéramy, Pierre-Emmanuel Hladik, Anne-Marie Déplanche, Sébastien Dubé, *Simulation of Real-Time Scheduling with Various Execution Time Models*, 9th IEEE International Symposium on Industrial Embedded Systems (SIES), Jun 2014, Pise, Italy.

4.4 YARTISS

Pour ce simulateur tout comme pour le projet SimSo, code source, article [14] et documentation sont disponibles. Nous apprenons donc que celui-ci permet d'évaluer et comparer des politiques d'ordonnancement sur des systèmes multiprocesseurs. Il se distingue principalement par l'intégration de modèles énergétiques. Il est développé en Java et fait le choix d'intégrer la génération des tâches ainsi que l'analyse des résultats directement au sein du logiciel, en plus de pouvoir utiliser des outils externes. Il est également en mesure de lancer de multiples simulations et de générer des traces concernant plusieurs métriques [15]. Le code source en lui-même étant peu documenté, peu d'information ont été obtenues concernant la logique permettant de passer d'évènements en évènements mais tout semble indiquer que celui-ci n'applique pas la procédure que nous souhaiterions appliquer afin d'avoir une diversité d'algorithmes et de systèmes supportés par la méthode d'élimination des temps creux.

14. [8] Y. Chandarli, F. Fauberteau, D. Masson, S. Midonnet, M. Qamhieh, et al. *Yartiss : A tool to visualize, test, compare and evaluate real-time scheduling algorithms*, 3rd International Workshop on Analysis Tools and Methodologies for Embedded and Real-time Systems, 2012, p. 21–26.

15. e.g préemptions, périodes d'inactivité, etc.

4.5 Conclusion

Au sein de ces quatre simulateurs, aucun ne semble appliquer de calcul spécifique généraliste pour effectuer un « saut temporel [16] » vers le prochain évènement tel que nous voulons l'appliquer à notre cas d'étude Cheddar. Comme nous avons pu le voir, certains projets abordent la problématique event-driven d'une manière intéressante. Cependant nous voyons qu'aucun de ces projets n'est vraiment généraliste en terme d'algorithmes et de systèmes supportés et que la majorité d'entre eux se basent sur des modèles spécifiques à la problématique à laquelle ils essayent de répondre [17]. Or, cela limite grandement les possibilités et l'intérêt recherché étant donné que Cheddar permet d'implémenter ses propres politiques d'ordonnancement et supporte déjà nativement une vingtaine d'algorithmes d'ordonnancement. Hors de ces projets liés à notre problématique, un framework généraliste basé sur la modélisations de systèmes à structure variable [18] et destiné à la simulation event-driven a été étudié : Adevs. Ce framework programmé en C++ permet de définir simplement sa simulation, étant donné que l'ensemble des fonctionnalités de simulation sont déjà implémentées. Or, la fonctionnalité que nous recherchons est de façon naïve, « *comment passer de l'unité de temps x à l'unité future y pour éviter les temps creux entre ces deux instants* », et doit s'inscrire dans le cadre de la simulation d'ordonnancement avec toutes les contraintes apportées par le domaine. Ainsi, la fonction gérant ce saut temporel au sein de la simulation n'est pas programmée et la problématique reste donc la même quand à savoir de quelle manière nous pouvons appliquer de manière généraliste une politique event-driven à notre cas d'étude. Cependant, cette partie de documentation nous a permis de confirmer le fait qu'aucun travaux ne semble avoir été effectué permettant de répondre de façon spécifique à notre problématique.

16. Nous entendons par saut temporel la possibilité pour le simulateur de changer son état courant en modifiant l'unité de temps courante vers une unité de temps future e.g incrémenter la valeur itérative de la boucle de simulation.

17. e.g YARTISS pour l'évaluation de la consommation énergétique en fonction des algorithmes d'ordonnancements.

18. [9] A. M. Uhrmacher. *Dynamic structures in modeling and simulation : a reflective approach, ACM Transactions on Modeling and Computer Simulation*, Vol. 11, No. 2 , April 2001, p. 206-232.

5 Approche Théorique

Dans cette partie, nous allons aborder la manière que nous avons choisi pour résoudre notre problématique principale. Comme nous l'avons vu dans la présentation de la méthodologie en section 3.3, résumant les différentes parties de ce rapport, nous avons convenu entre les différents intervenant de ce stage, de mettre en place un système d'heuristiques. En effet, comme nous le verrons dans la prochaine sous-partie sur la structuration du problème, les contraintes posées par l'optimisation d'un simulateur time-driven tel que Cheddar afin de l'orienter vers une simulation événementielle sont importantes, et un système d'heuristique qui pourrait permettre d'y répondre. De plus, nous avons pu voir que dans la conclusion de la partie de documentation bibliographique en section 4.5 qu'aucun des travaux étudiés n'a répondu à nos exigences.

Nous aborderons donc dans un premier temps la structuration du problème afin d'identifier vraiment de quelle manière nous pouvons le résoudre. Dans un second temps nous parlerons des intervalles creuses puis la définition d'un ensemble d'heuristiques applicables à ces intervalles dites « creuses ». Enfin, nous aborderons les aspects de complexité générés par la solution et les implications de celle-ci puis nous finirons cette partie concernant la construction de notre approche théorique par une proposition d'architecture pour implémentation. Nous utiliserons la légende suivante au sein des chronogrammes :

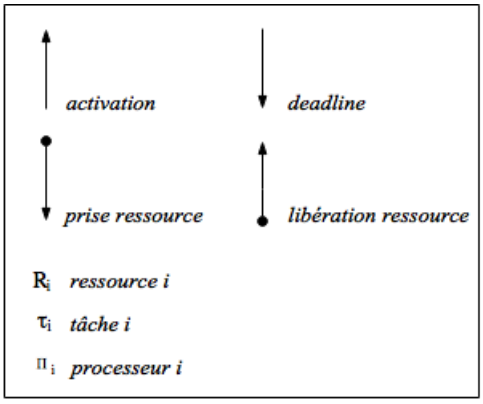

FIGURE 1 – Légende des chronogrammes.

5.1 Structuration du problème

À première vue, toute modification de l'unité de temps courante vers une une unité de temps future (saut temporel) au sein de ce que nous appellerons intervalle instable entraînera une perturbation de l'ordonnancement. En effet, cette intervalle instable peut-être caractérisée comme l'intervalle où entre les deux bornes de temps qui la composent, un saut temporel perturbera l'ordonnancement du système. En conséquent, nous pouvons en déduire que ces deux bornes dépendent intrinsèquement du système considéré mais surtout de l'algorithme d'ordonnancement utilisé.

En effet, pour un système donné, son ordonnancement sera différent en fonction de l'ordonnanceur utilisé. C'est donc l'ordonnanceur et donc son algorithme qui dictent indirectement le placement de ces bornes.

Un des points majeurs à prendre en considération dans l'optimisation du temps de simulation et donc dans la définition de cette intervalle instable, est la manière dont on souhaite optimiser la simulation elle-même. En effet, l'heuristique que l'on souhaite appliquer joue sur les bornes de l'intervalle instable. Ainsi, si l'on souhaite par exemple optimiser la simulation de sorte à éviter la production et le calcul d'évènements lorsque aucune tâche n'est en cours d'exécution sur le ou les processeurs, nous pouvons rapidement en déduire une intervalle stable de saut temporel possible où la borne minimum est le moment où la dernière tâche actuellement en cours d'exécution termine sa capacité et la borne maximum étant celle ou se produit le réveil de la prochaine tâche prévue dans l'ordonnancement.

L'intervalle instable est donc l'opposée [19] de cette intervalle stable, comme nous pouvons le voir sur le schéma suivant :

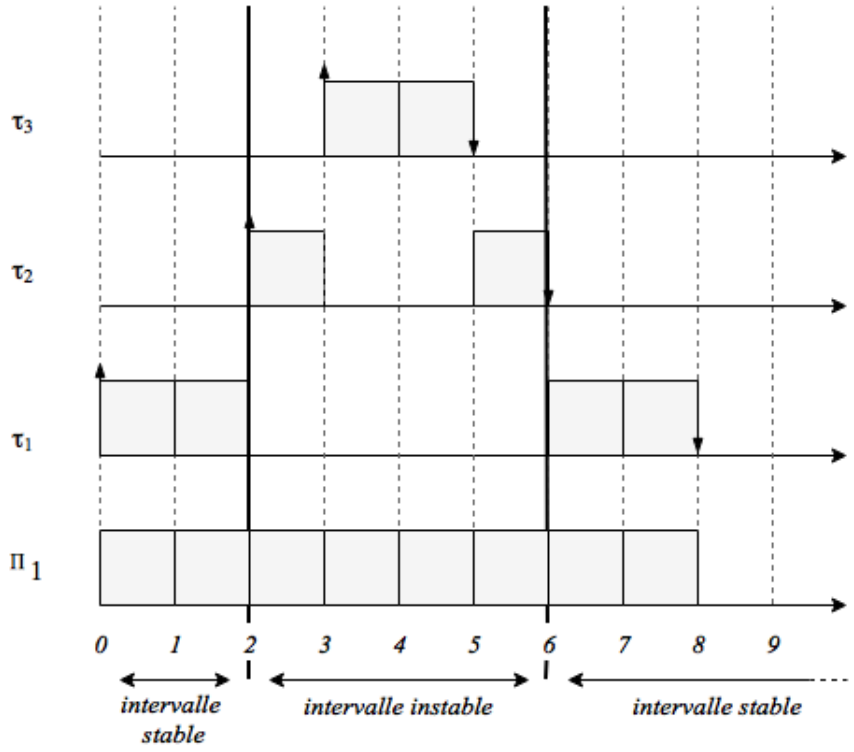

FIGURE 2 – Chronogramme d'ordonnancement avec caractérisation des intervalles stables et instables.

Nous voyons donc d'ors et déjà que l'heuristique définit de quelle manière seront placées ces bornes sur le chronogramme. L'algorithme d'ordonnancement, lui, influe sur les unités de temps où seront placées ces bornes puisqu'il régit l'ordonnancement en lui-même et donc le placement des tâches. En imaginant une intervalle instable pour un système donné et dont le placement des bornes est inchangeant peu importe l'ordonnanceur utilisé, on peut en déduire que hors de cette intervalle le comportement du système sera donc semblable. En effet, cette intervalle est définie de manière

19. Dans ce cas présent, l'intervalle stable est composée exclusivement de temps creux et l'intervalle instable correspond à la période où aucun temps creux n'est présent (période d'activité).

à éviter toute instabilité qui pourrait conduire à un comportement non prédictif [20] de manière simple. Ainsi, en reprenant l'heuristique de la Figure 2, nous pouvons voir que hors de cette intervalle, aucune préemption ne peut se produire. Cependant, une tâche apériodique peut-être réveillée dans cette intervalle, point que nous aborderons par la suite.

Sans faire cas de cette problématique des tâches apériodiques, nous voyons donc que la définition de l'heuristique ajoute la gestion de certains éléments que l'on peut qualifier de perturbateurs [21] en fonction de celle-ci. De plus, l'heuristique et ce qu'elle cherche à optimiser influe sur le nombre de calculs à effectuer. Une heuristique très précise et très complexe produira un grand nombre de calculs afin d'obtenir les bornes de l'intervalle stable et très probablement pour une optimisation minime. Une heuristique simple risque donc de garantir le plus de gains en terme de calculs mais surtout en terme de production du nombre d'évènements et donc du nombre d'unités de temps qui seront supprimées de l'ordonnancement.

Si l'on prend une définition naïve d'une simulation event-driven et en considérant comme exemple une queue, un évènement sera l'arrivée ou le départ d'une personne dans la queue. Ainsi, aucun calcul n'est effectué entre chacun des évènements. Or dans l'intervalle instable, nous pouvons dire que nous avons une personne qui peut arriver et qui peut partir à chaque unité de temps. Nous devons donc effectuer les calculs et donc les vérifications à chaque unité de temps puisque dans un ordonnancement, l'état courant dépend des états précédents. Étant donné le nombre d'algorithmes présents dans Cheddar où nous nous devons d'implémenter notre solution d'optimisation, l'heuristique semble être la meilleure solution tant qu'elle reste généraliste et le plus simple possible dans son calcul.

En conclusion, nous pouvons dire que nous cherchons à obtenir une intervalle sur les évènements où il n'est pas possible d'effectuer un saut temporel sans garantie que l'ordonnancement ne soit pas modifié. Si cette exigence est garantie, cela nous permet donc d'obtenir les temps où aucun évènement d'intérêt [22] est présent.

20. e.g une préemption.
21. e.g des dépendances entres les tâches, la préemptivité, l'accès commun à une ressource etc.
22. Nous pouvons définir un évènement d'intérêt comme étant essentiel à la compréhension de l'ordonnancement.

5.2 Intervalles creuses

En premier lieu, et comme nous l'avons vu dans la définition de la problématique principale, nous souhaitons définir des heuristiques qui ciblent des intervalles possédant un motif spécifique. Comme nous l'avons vu, ce motif peut cibler les temps d'inactivité du processeur et nous pouvons l'étendre à d'autres cas tant que le calcul de l'heuristique reste simple en terme de complexité mais surtout qu'il évite les instabilités du système que nous avons décrits plus avant.

On peut donc définir une intervalle de temps creux, ou du moins de temps sans instabilité, comme étant une intervalle au sein de laquelle nous éviterons des évènements telles que les préemptions. Nous pouvons donc prendre par exemple le cas d'un modèle où à un temps t, une tâche τ_i est la dernière en exécution. Nous pouvons donc assurer de façon sûre que tant qu'une autre tâche n'est pas activée avant la fin de l'exécution de notre tâche, celle-ci ne risque aucune préemption et nous pouvons donc être sûr que l'ordonnanceur laissera la tâche terminer sa capacité.

Nous pouvons aussi appliquer une heuristique sur les systèmes où est appliqué un algorithme d'ordonnancement non-préemptif. Les tâches ne sont donc pas préemptées et l'exécution est simple à déterminer. Nous pouvons donc effectuer un saut temporel à partir du temps de démarrage d'un travail au temps de fin de capacité de ce travail et ainsi de suite pour chaque tâche du système. Cependant, il faut adapter la génération de telle sorte à ne pas perdre l'information des évènements d'exécution du travail sauté. Cette problématique est d'ordre sémantique au niveau de la sortie textuelle du simulateur au sein de sa table d'évènements.

Dans le cas ou nous considérons un système possédant un ordonnanceur non-préemptif, nous pouvons déterminer qu'aucune instabilité ne se produira car une tâche s'exécutera sans interruption, de sa date de début d'exécution jusqu'à sa date de fin de consommation de capacité. Ainsi, seuls persisterons les évènements de début d'exécution, de fin d'exécution, et les évènements relatifs aux accès aux ressources et dépendences propres à cette tâche.

Maintenant que nous avons défini les optimisations que nous cherchons à obtenir et de quelles manières nous pouvons les effectuer, nous allons décrire de façon plus détaillée les différentes heuristiques considérées, leurs structure et leurs limitations.

5.3 Définition des heuristiques

La problématique et les pistes de résolutions via l'utilisation d'heuristiques ont désormais été introduites et nous pouvons dorénavant développer dans cette section les différentes heuristiques considérées.

5.3.1 Temps creux

Nous avons vu dans la structuration du problème d'optimisation un exemple d'heuristique basée sur les temps creux du système. Un temps creux peut-être définit comme suit :

"Un temps creux peut-être défini comme étant une unité de temps au sein de la simulation durant laquelle aucun évènement n'est produit."

Nous avons donc besoin pour cette heuristique de déterminer une intervalle de temps où la borne minimum est l'unité de temps t^- à partir de laquelle la dernière tâche en cours d'exécution consomme la totalité de sa capacité, et la borne maximum l'unité de temps t^+ à partir de laquelle la tâche ayant son réveil au plus tôt est activée.

Nous pouvons ainsi définir comme suit l'intervalle S représentant notre intervalle stable au sein de l'ordonnancement : $S = [t^-, t^+]$.

Plus les périodes sont grandes et les capacités faibles par rapport aux périodes respectives, plus les bénéfices en terme d'optimisation temporelle et donc génération d'évènements sont importants. En effet, en regardant l'exemple suivant nous voyons que nous économisons 65% du temps d'exécution sur l'hyperpériode.

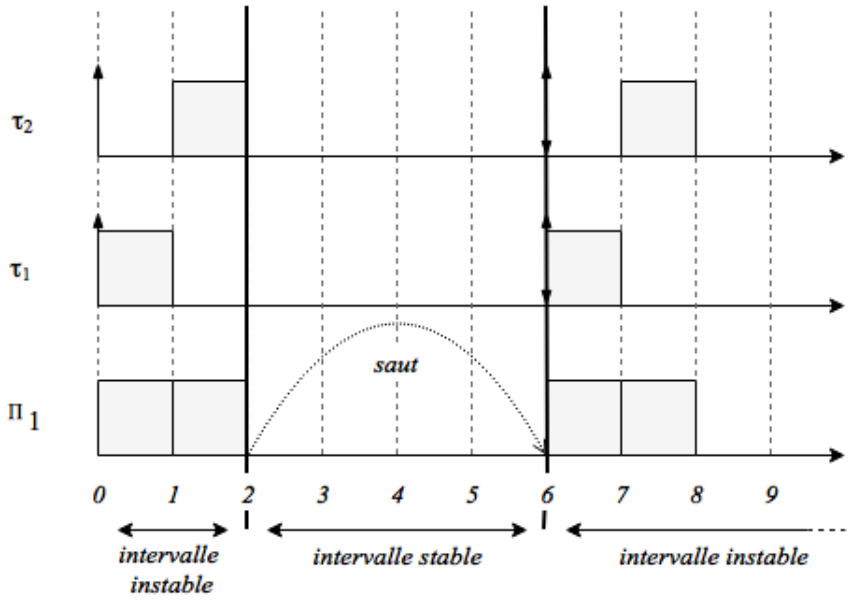

FIGURE 3 – Chronogramme d'ordonnancement, unités de temps optimisées.

Nous pouvons voir que si nous avons au sein du modèle de tâches des tâches sur des dimensions temporelles différentes, e.g s, μs, les bénéfices sont très important puisque une large partie du temps de la simulation risque d'être dédiée aux temps creux. En effet, la boucle de simulation effectue ses itérations sur la dimension temporelle la plus petite. En fonction de l'écart de dimensions temporelles on peut se retrouver avec des ratios très importants en terme de capacité e.g 1/1000.

Si nous voyons les bénéfices possibles en évitant ces temps creux, il faut aussi en percevoir les limitations. En effet, sur un système où le taux d'utilisation des unités d'exécution est élevé, l'heuristique deviens caduc et l'est totalement sur un taux d'utilisation processeur à 100% comme nous pouvons le voir sur la Figure 4 ci-après.

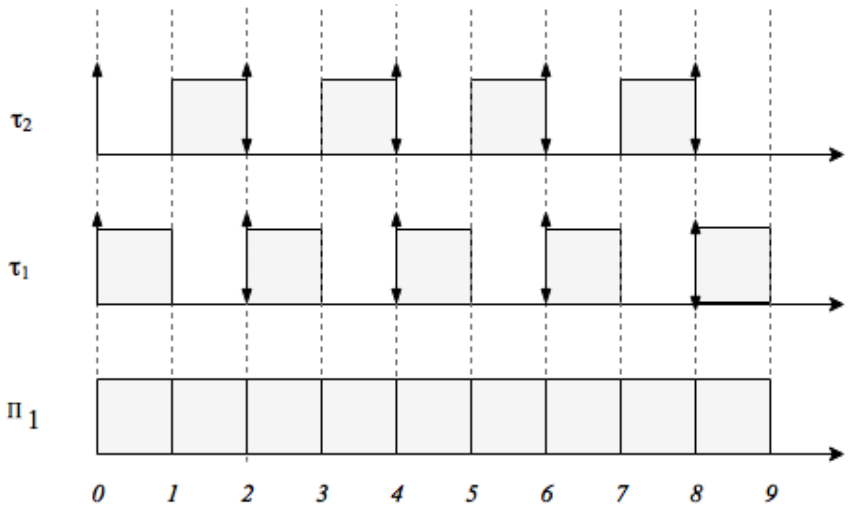

FIGURE 4 – Chronogramme d'ordonnancement sans temps creux.

Nous pouvons donc désormais en définir l'algorithme 1 ci-dessous, au plus simple possible, que nous complexifieront lors de l'implémentation en fonction des contraintes du support sur lequel celui-ci sera intégré.

Algorithme 1 : Évitement des temps creux

```
Result : TempsCourant est mis à jour
Initialisations() ;
if Aucune tâche active then
    for tâche i de 0 à nombre de tâches - 1 do
        if Tâche(i).TempsDeRéveil < PlusProcheRéveil then
            PlusProcheRéveil := Tâche(i).TempsDeRéveil ;
        end
    end
    TempsFutur := PlusProcheReveil ;
    if TempsFutur > TempsCourant then
        TempsCourant := TempsFutur ;
    end
end
```

Il serait possible de calculer un seuil au-delà duquel l'heuristique basée sur les temps creux serait par défaut désactivée car contre productive. En effet, si il est possible de calculer le taux d'utilisation du processeur, et en fonction de ce taux en définir l'efficacité de l'algorithme, nous pourrions donc en déduire un seuil au-delà duquel l'heuristique n'est plus efficace.

5.3.2 Dernier travail actif

En considérant un système possédant une tâche unique, nous pouvons considérer qu'aucune instabilité majeure se produira. En effet, la tâche étant seule, l'ordonnanceur même si préemptif ne préemptera pas la tâche puisque aucune charge de travail concurrente ne doit être exécutée. Nous pouvons donc considérer que le système est stable dans la mesure où les évènements relatifs à cette tâche sont prédictibles.

En effet, le modèle Cheddar considère par exemple pour les accès à une ressource que la prise de ressource est à l'unité t_i de sa capacité et la libération à l'unité t_f. Ainsi, même si nous avons une tâche qui génère des évènements supplémentaires de part les actions quelle effectue, les informations nécessaires nous sont disponibles. En effet, nous pouvons voir sur la page suivante dans le Listing 2, l'extrait d'un modèle XML pour Cheddar contenant les informations concernant la section critique et à quels moments de sa capacité une tâche accède à une ressource.

```
<resources>
  <pcp_resource id="root.my_platform.cpu.my_process.d1">
    <name>root.my_platform.cpu.my_process.d1</name>
    <object_type>RESOURCE_OBJECT_TYPE</object_type>
    <protocol>PRIORITY_CEILING_PROTOCOL</protocol>
    <address_space_name>
      root.my_platform.cpu.my_process
    </address_space_name>
    <cpu_name>root.my_platform.cpu</cpu_name>
    <state>1</state>
    <size>0</size>
    <address>0</address>
    <critical_sections>
      <task_name>root.my_platform.cpu.my_process.t1</task_name>
      <critical_section>
        <task_begin>1</task_begin>
        <task_end>8</task_end>
      </critical_section>
      <task_name>root.my_platform.cpu.my_process.t2</task_name>
      <critical_section>
        <task_begin>5</task_begin>
        <task_end>10</task_end>
      </critical_section>
    </critical_sections>
  </pcp_resource>
</resources>
```

Listing 2: Définition XML d'un accès ressource d'un modèle pour Cheddar

Nous cherchons donc à obtenir une heuristique où la borne minimum de l'intervalle de saut est le moment où la tâche commence son exécution et la borne maximum étant le prochain évènement lié à la tâche[23]. Pour une tâche effectuant un accès à une ressource, nous aurons donc deux intervalles de saut au minimum. En effet, nous aurons les intervalles de saut suivantes : [*Début d'exécution ; Début accès ressource*], [*Fin accès ressource ; Fin d'exécution*] ou [*Fin accès ressource ; Début accès*

23. e.g fin de capacité, prise de ressource etc.

ressource] [24].

Nous pourrions aussi ajouter une logique de saut supplémentaire au sein de l'action qu'effectue la tâche, ici [*Début accès ressource ; Fin accès ressource*], cependant comme nous avons plusieurs actions possibles lors de l'exécution d'une tâche cela risque de demander la définition d'un nombre important de conditions. De plus, ce sont des intervalles qui sont généralement de longueur faible et le gain risque d'être quasi-nul [25].

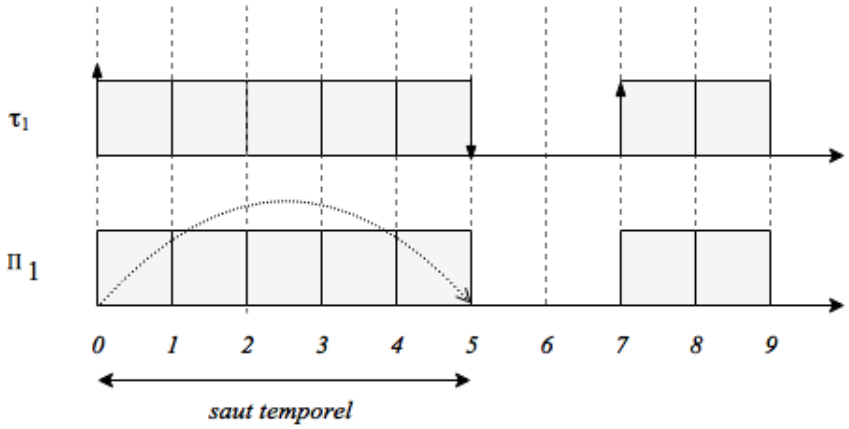

FIGURE 5 – Chronogramme d'ordonnancement, tâche seule sans évènement supplémentaire.

24. Le motif interne d'accès à une ressource peut se répéter.

25. e.g un accès à une section critique est en général assez limité dans le temps en rapport avec les autres traitements que pourrait effectuer une tâche. L'utilité d'optimiser ces intervalles internes à l'exécution d'une tâche est donc discutable.

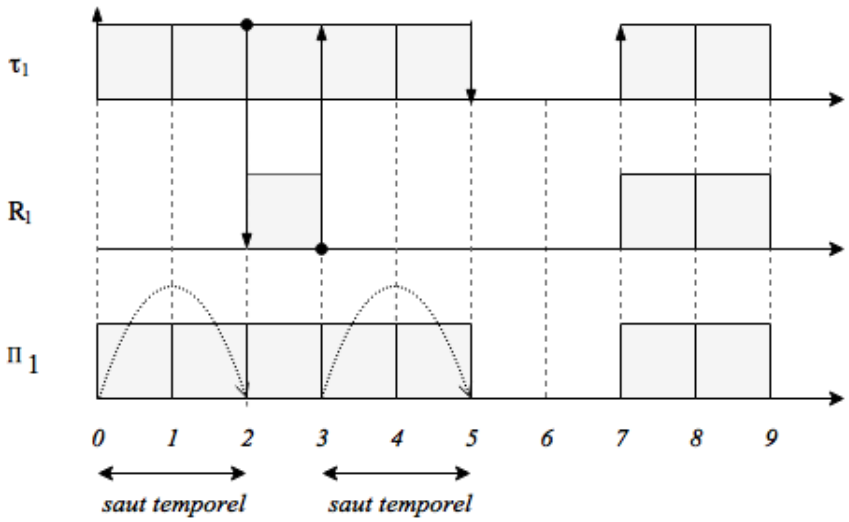

FIGURE 6 – Chronogramme d'ordonnancement, tâche seule avec accès ressource.

Le système actuellement considéré est un système possédant une tâche unique et ne représente pas d'intérêt majeur [26]. Cependant, il est possible d'étendre l'heuristique au systèmes multi-tâches. En effet, la borne minimum initiale de cette heuristique sera donc la date de début d'exécution d'une tâche τ_i lorsque toutes les autres tâches ont terminées leur exécution pour leur travail courant soit le moment où cette tâche deviens seule en exécution : [moment où tâche seule ; prochain évènement lié à la tâche].

26. Un système ne possédant qu'une seule tâche ne représente bien évidemment pas d'intérêt dans le cadre d'une simulation d'ordonnancement car aucune politique d'ordonnancement ne sera appliquée.

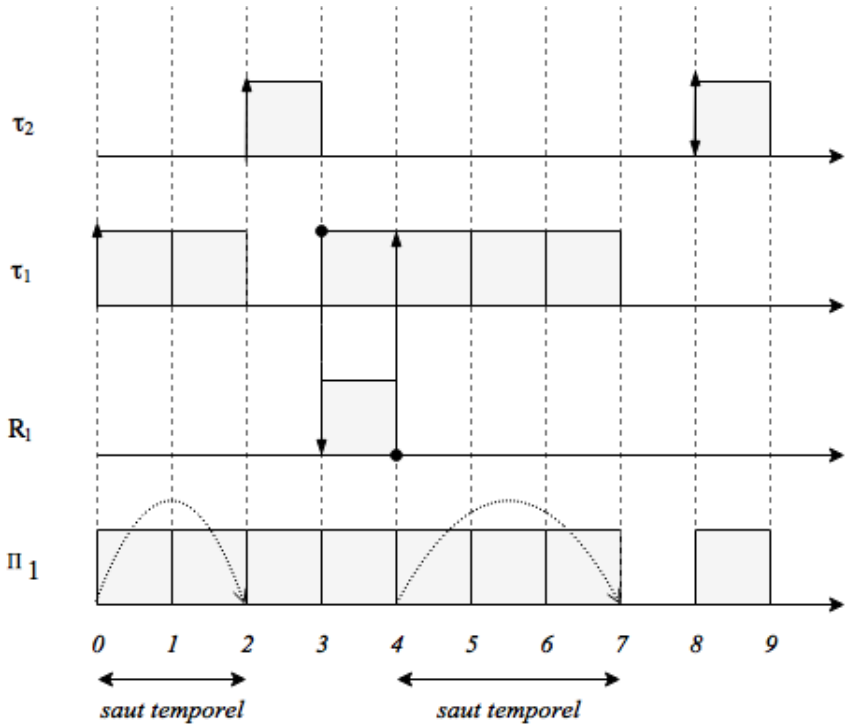

FIGURE 7 – Chronogramme d'ordonnancement, tâches multiples avec accès ressource et capacité entamée.

Nous voyons donc en Figure 7 que l'heuristique est possiblement applicable à l'ensemble des systèmes avec une efficacité variable en fonction du nombre d'unités de temps ou un travail est seul en exécution sur l'intervalle définie.

Nous pouvons donc définir l'algorithme suivant :

Algorithme 2 : Évitement des unités de temps - dernier travail actif sur l'intervalle ciblée

```
Result : TempsCourant est mis à jour
Initialisations() ;
if une seule tâche active then
    for tâche i de 0 à nombre de tâches - 1 do
        if Tâche(i).TempsDeRéveil < PlusProcheRéveil then
            PlusProcheRéveil := Tâche(i).TempsDeRéveil
        end
    end
    if PlusProcheRéveil < (TempsCourant + TâcheActive.ResteDeCapacité)
     then
        TempsFutur := PlusProcheRéveil ;
        if TâcheActive accède ressource and TâcheActive n'est pas en prise de
         ressource then
            for ressources r accédées par TâcheActive do
                if TâcheActive.DatePriseRessource(r) > TempsCourant then
                    if TâcheActive.DatePriseRessource(r) < TempsFutur then
                        TempsFutur := TâcheActive.DatePriseRessource(r) ;
                    end
                end
            end
        end
    end
    TempsCourant := TempsFutur ;
end
```

Sur la Figure 8 suivante nous pouvons voir une optimisation du nombre d'unités de temps de 50%.

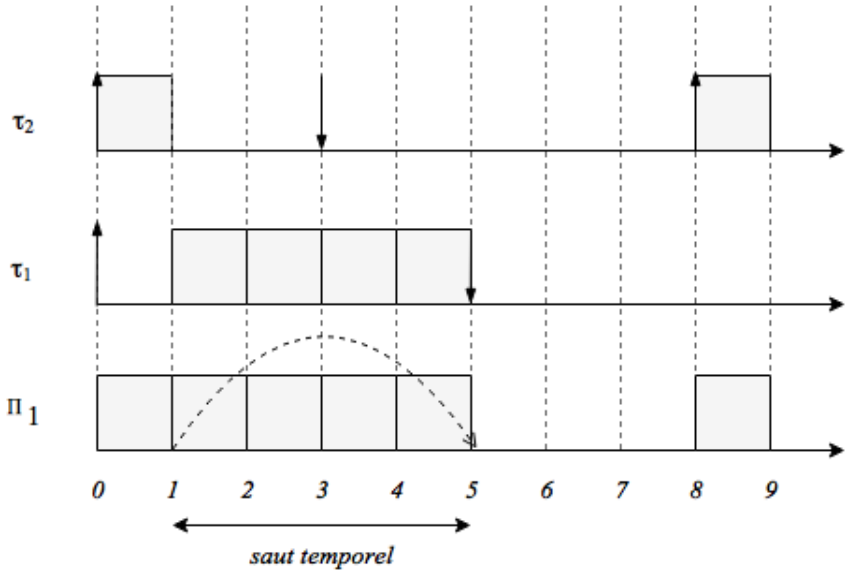

FIGURE 8 – Chronogramme d'ordonnancement, exemple d'application de l'heuristique 2

En combinant avec la première heuristique nous obtenons sur la Figure 9 une optimisation du nombre d'évènements de l'ordre de 75% :

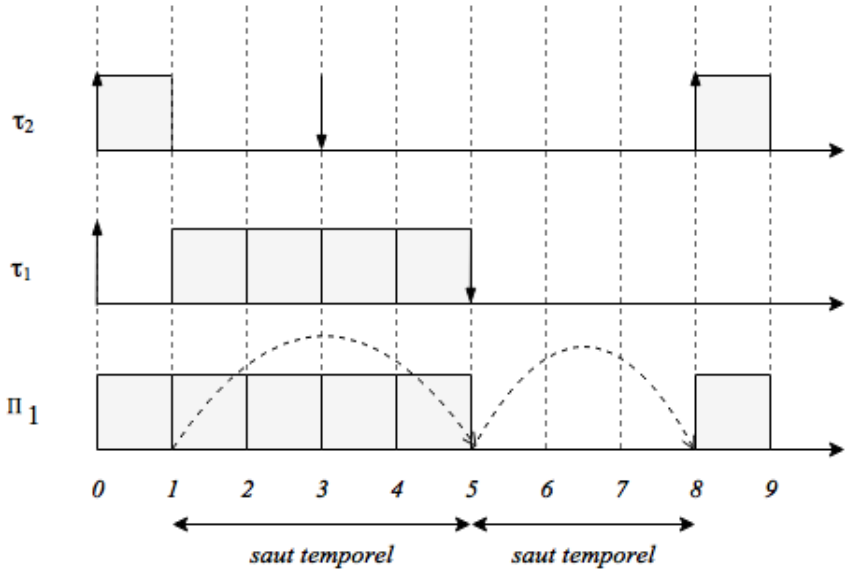

FIGURE 9 – Chronogramme d'ordonnancement, exemple d'application des deux premières heuristiques.

Nous avons dorénavant définit la portée de l'heuristique, son fonctionnement ainsi qu'une estimation de ses performances. Nous allons donc dans le point suivant étendre son fonctionnement pour les systèmes à ordonnanceur non-préemptif.

5.3.3 Non-préemptivité

Si nous considérons un système avec ordonnanceur non-préemptif, aucune instabilité liée à l'acte de préemption d'une tâche n'est possible. Cela nous permet donc d'étendre possiblement l'heuristique précédente pour l'ensemble des tâches du système. En effet, l'ordonnancement étant non-préemptif, lorsqu'une tâche entame sa capacité, celle-ci la consomme sans interruption jusqu'à sa dernière unité. Nous avons donc de multiples tâches mais celle-ci peuvent être considérées indépendantes des unes des autres dans la mesure ou aucune préemption de peut se produire. Cependant, cela n'empêche pas l'existence de dépendances diverses entre les tâches.

Nous cherchons donc à obtenir le même fonctionnement que l'heuristique précédente mais étendu à l'ensemble des tâches du système. Les bornes minimum et maximum restent les mêmes mais s'appliquent à chaque tâches. Ainsi, nous aurons les intervalles de saut suivantes : [*Début d'exécution ; Début accès ressource*], [*Fin accès ressource, Fin d'exécution*]. Nous pourrions aussi ajouter une logique de saut supplémentaire au sein de l'action qu'effectue la tâche, par exemple [*Début accès ressource ; Fin accès ressource*]. Cependant comme nous avons plusieurs actions possibles lors de l'exécution d'une tâche cela risque de demander la définition d'un nombre important de conditions comme nous l'avons déjà vu dans l'heuristique précédente. Une des intervalle ne sera pas présente dans cette heuristique. En effet, l'intervalle [*moment ou tâche seule ; prochain réveil de tâche*] ne se produira jamais puisque les tâches ne peuvent pas être interrompues.

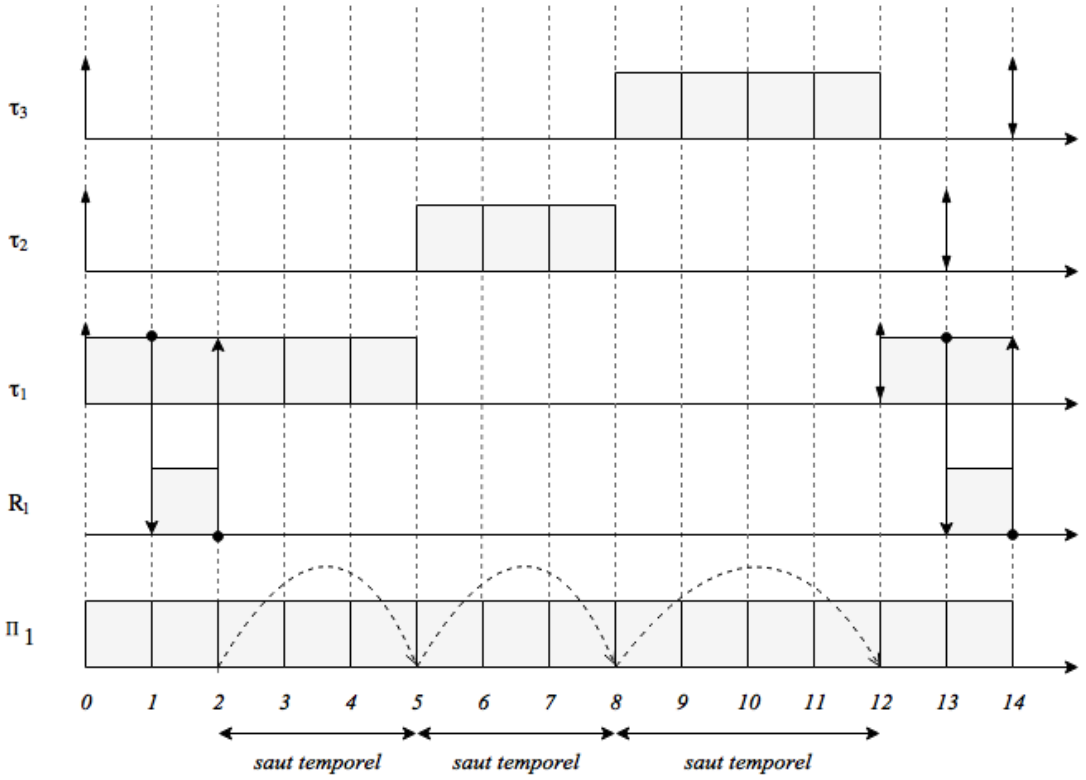

FIGURE 10 – Chronogramme d'ordonnancement, exemple d'application de l'heuristique 3 pour systèmes non-préemptif.

L'algorithme peut-être défini comme suit :

Algorithme 3 : Évitement des unités de temps - système non-préemptif

```
Result : TempsCourant est mis à jour
Initialisations() ;
if système non-préemptif then
    TempsFutur := TempsCourant + TâcheActive.ResteDeCapacité ;
    /* conditions et traitements de dépendances                    */
    /* e.g prise de ressource                                      */
    if TâcheActive accède ressource and TâcheActive n'est pas en prise de
     ressource then
        for ressources r accédées par TâcheActive do
            if TâcheActive.DatePriseRessource(r) > TempsCourant then
                if TâcheActive.DatePriseRessource(r) < TempsFutur then
                    TempsFutur := TâcheActive.DatePriseRessource(r) ;
                end
            end
        end
    end
    TempsCourant := TempsFutur ;
end
```

Le gain dans le cas d'une ordonnanceur non-préemptif risque d'être important. En effet, sur le chronogramme précédent nous pourrions obtenir une diminution du nombre d'évènements de $\frac{7}{12}$.

Maintenant que nous avons défini la dernière heuristique que nous considérerons, nous pouvons aborder le cas d'autres heuristiques et les problématiques de complexité qui leurs sont liées.

5.4 Complexité et implications

Une des problématique majeure de l'utilisation d'heuristiques diverses est la complexité qui leur est associée. En effet, il faut maintenir les heuristiques au plus simple dans leur comportement et conditions sous peine d'explosion de leur complexité. En effet, plus le nombre de conditions est important, plus le nombre de calculs et de données nécessaires et à traiter sont importants. Le risque est donc d'avoir une heuristique qui coûte plus en terme de calculs que le fonctionnement initial de la simulation. De plus, les implications des algorithmes d'ordonnancements utilisés dans la pratique tel les algorithmes hiérarchiques ne peuvent pas de façon correcte être définis de manière théorique. En effet, ce type de contraintes sera résolu lors de l'implémentation, là où les heuristiques seront adaptées en fonction des contraintes du simulateur d'ordonnancement et de son architecture.

Une autre problématique est la multiplicité des heuristiques. En effet, il est peut-être possible de définir une heuristique unique qui englobe la totalité des conditions et possibilités de saut temporel. Cependant, si une telle heuristique existe, elle sera soumise elle-aussi à la problématique de complexité et devra donc avoir une complexité inférieure à la simulation initiale afin d'être valable.

Il faut de plus garder à l'esprit que dans le cadre du stage et sa durée limitée, l'objectif n'était non pas de trouver toutes les heuristiques possibles et de les implémenter mais bien d'étudier la possibilité de l'utilisation d'heuristiques et l'implémentation de ces quelques heuristiques afin de confirmer ou d'infirmer les gains en terme de temps et de nombre d'évènements générés.

6 Réalisation

Nous aborderons dans cette partie l'étape de réalisation. Cette étape englobe la description et l'analyse des outils sur lesquels seront implémentés diverses fonctionnalités ainsi que le système d'heuristiques défini dans la partie précédente. Cheddar servira de cas d'étude et de test de l'implémentation de la solution et Marzhin servira de cas pratique d'implémentation de la solution proposée pour Cheddar au sein d'un autre simulateur existant et ayant un fonctionnement interne de simulation radicalement différent.

6.1 Présentation de Cheddar

Nous avons pu aborder au sein de la partie documentation un receuil d'écrits portant sur le sujet de la simulation event-driven et plus particulièrement dans le cas qui nous intéresse, appliqué au domaine de la simulation d'ordonnancement temps-réel. La partie qui a été effectuée après cette recherche de documentation est donc l'analyse de l'existant. En effet, c'est le premier point après la recherche de documentation qui va nous permettre de comprendre les tenants et aboutissants des supports sur lesquels les solutions développées seront apportées. Je vais présenter dans cette partie trois logiciels sur lesquels mon travail m'a porté à intervenir avec des niveaux d'implication et de modifications variable.

Les sous-parties suivantes résument les informations sur Cheddar, son architecture et son fonctionnement tel que je peux les résumer à partir de l'analyse, des travaux et des informations disponibles sur son sujet.

6.1.1 Description

Cheddar est un logiciel open-source distribué sous licence *GNU General Public License (GPL)* développé en langage Ada et doté d'une interface graphique basée sur la librairie graphique GtkAda. Modulaire dans ses fonctionnalités et dotés de multiples outils, son existence est motivée par la création d'un outil permettant l'étude des systèmes temps réels. Il s'inscrit dans différents cadres dont les principaux sont sa vocation à servir comme support de recherche, d'outil à vocation pédagogique [27] et comme outil de prototypage de systèmes temps réels.

Celui-ci fourni un ensemble de fonctionnalités dont la principale est la vérification d'ordonnançabilité. En effet, celui-ci permet cette vérification de modèles par deux vecteurs. Le premier est par l'utilisation de tests de faisabilités mais ne couvrant pas l'ensemble des algorithmes d'ordonnancement ni l'ensemble des architectures. Le second est la vérification par simulation qui étend le champ d'application des tests de vérification en permettant ainsi d'exposer le comportement du modèle sur une durée définie par l'utilisateur sur tout les algorithmes d'ordonnancement implémentés au sein de Cheddar mais aussi sur des politiques d'ordonnancement définies par l'utilisateur. Il permet aussi de définir son système temps réel à soumettre à l'analyse grâce à un éditeur graphique ainsi que de nombreuses autres fonctionnalités.

27. [10] Frank Singhoff, Alain Plantec, Stéphane Rubini, Hai-Nam Tran, Vincent Gaudel, et al.. *Teaching Real-Time Scheduling Analysis with Cheddar*, 9ème édition de l'Ecole d'Eté « Temps Réel », Aug 2015, Rennes, France.

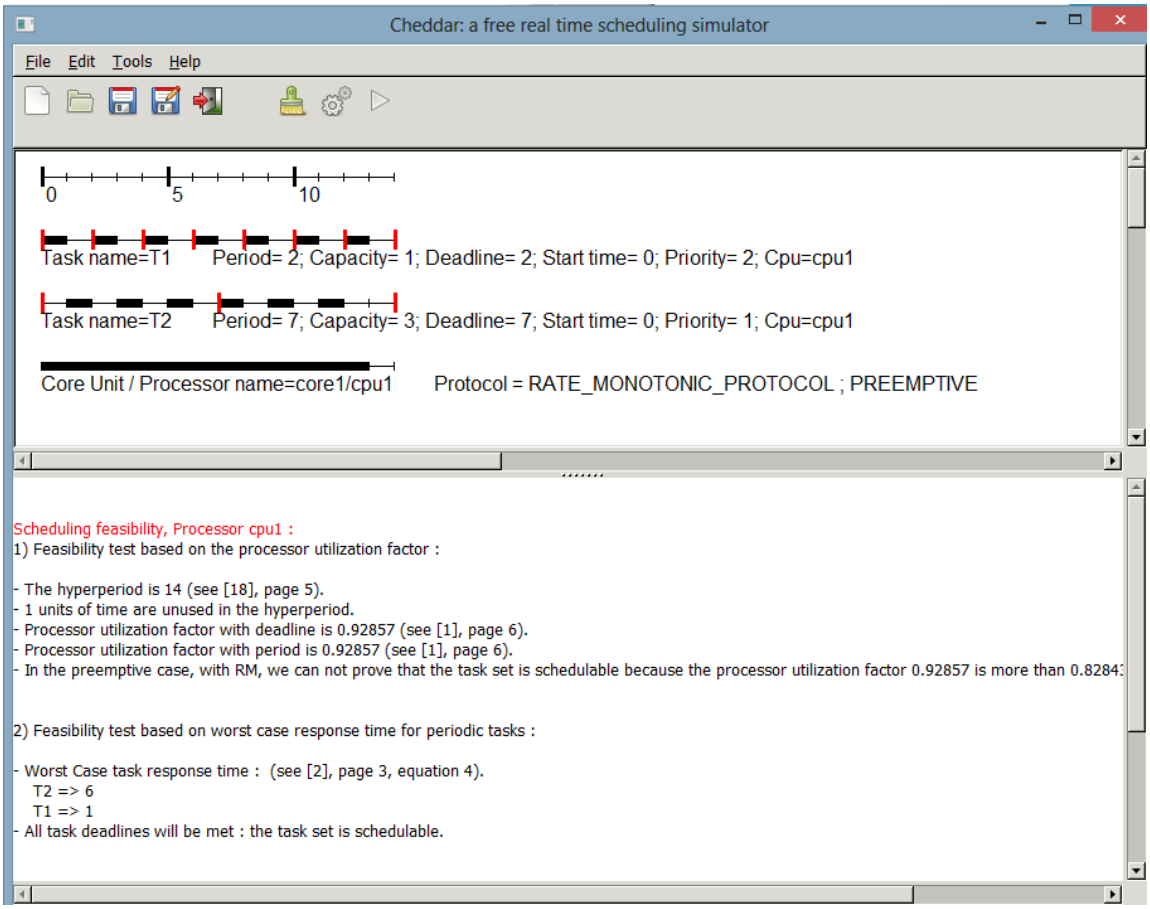

FIGURE 11 – Interface graphique du simulateur Cheddar.

L'entreprise Ellidiss est comme nous l'avons vu, partie prenante du développement de l'outil. Cela est en partie motivé par le fait que celui-ci soit utilisé au sein d'un de leur logiciel nommé AADLInspector que nous décrirons plus après. Le système d'heuristiques que nous aborderons dans la partie qui lui est consacré devra donc être implémenté, comme nous l'avons vu dans la définition de notre approche théorique, au sein de l'outil Cheddar.

Nous cherchons donc dans cette partie à obtenir des informations sur la partie simulation de l'outil afin de connaître la manière dont les informations de simulation sont extraites, quelles informations sont disponibles ainsi que leurs translation dans le code logiciel.

En effet, c'est sur cette partie du logiciel que nous interviendrons afin d'essayer d'appliquer les bénéfices d'une simulation event-driven et donc de gagner du temps sur la simulation ainsi que bénéficier d'un réduction du nombre d'évènements générés.

6.1.2 Architecture

L'étude d'architecture va nous permettre de comprendre la structuration du logiciel, comment les différents modules se composent, comment le code est structuré et donc identifier les composants logiciels pouvant potentiellement être utiles à la réalisation du cahier des charges.

On cherche d'abord à obtenir une analyse à grande échelle nous permettant de visualiser de quelle manière les modules s'articulent les uns les autres. Cela va permettre dans le cycle de développement futur d'évaluer les zones ou sont susceptibles d'être présentes les fonctionnalités que nous recherchons et/ou de savoir où en implémenter de nouvelles. En effet, seule une connaissance de l'architecture va permettre de travailler sur le logiciel de façon efficiente. Si l'on s'attarde sur la structure des codes sources de l'outil nous pouvons observer une organisation en modules. En effet, les différents blocs de fonctionnalités sont divisés en sous dossiers. Ainsi, nous avons une organisation qui est visuellement efficace pour la compréhension afin d'identifier plus rapidement les fichiers pouvant contenir les fonctionnalités qui nous intéressent.

Ainsi, nous allons nous intéresser principalement au module `Scheduling_Simulation` qui contient les différents fichiers relatifs à la construction de la simulation, de la table d'évènements et autres, relatifs à la fonctionnalité de simulation d'ordonnancement. Afin d'effectuer la simulation d'ordonnancement, Cheddar effectue comme nous le verrons par la suite une suite de calculs afin de déterminer l'ordonnancement du système considéré. Cet ordonnancement est calculé et stocké au sein d'une table d'évènement. L'affichage se fait quant à lui une fois le calcul de la table d'évènement terminée.

Une des données importante à obtenir est la manière dont les différents appels de procédure sont effectués successivement afin de démarrer la simulation. Cela permet de comprendre de quelle manière les différentes couches de l'application communiquent entre elles afin d'effectuer la simulation. C'est une donnée nécessaire dans la mesure où il est possible d'intervenir sur des couches supérieures au module de simulation d'ordonnancement.

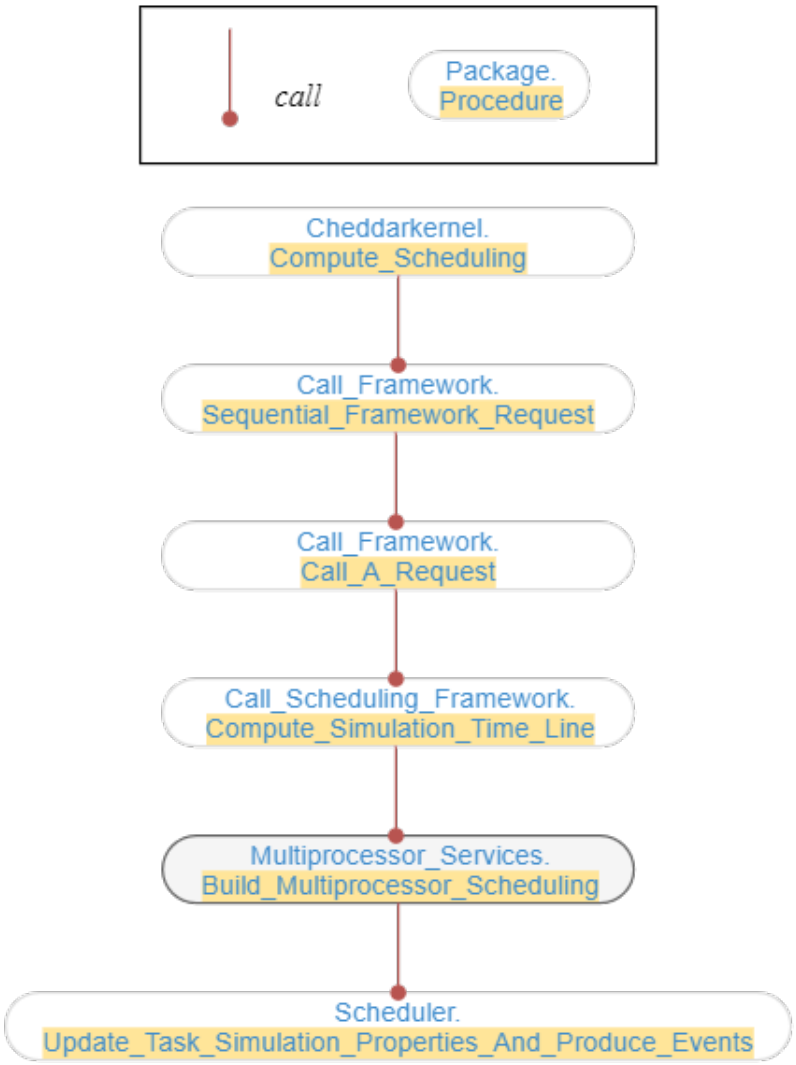

FIGURE 12 – Appels de fonction pour démarrage de la simulation d'ordonnancement.

Nous voyons donc sur la figure ci-dessus que la procédure `Build_Multiprocessor_Scheduling` est le coeur de la construction de la table d'évènements de la simulation d'ordonnancement puisque c'est la procédure faisant appel à la procédure de génération des évènements.

La simulation est centralisée. Ainsi, nous avons une gestion de l'ensemble des composantes et un fonctionnement global associé. Nous avons donc une boucle simple de simulation dont le nombre d'itérations dépend de la date de fin de simulation. De cette manière, une itération équivaut à une unité de temps. La boucle de simulation non modifiée est décrite en Annexe A.

Concernant les données de simulation, Cheddar modélise comme nous l'avons vu précédemment les différents composants de l'architecture au sein de son code mais pas leur fonctionnement. En conséquent, les données de simulations sont les instances des différents composants et sont rendus accessibles au sein de la partie simulation du logiciel via une structure de données spécifique. Cette structure de données contient les données de simulation de base des composants du système ainsi que des informations supplémentaires spécifiques à la simulation. Nous pouvons visualiser cette structure au sein de la Figure 13.

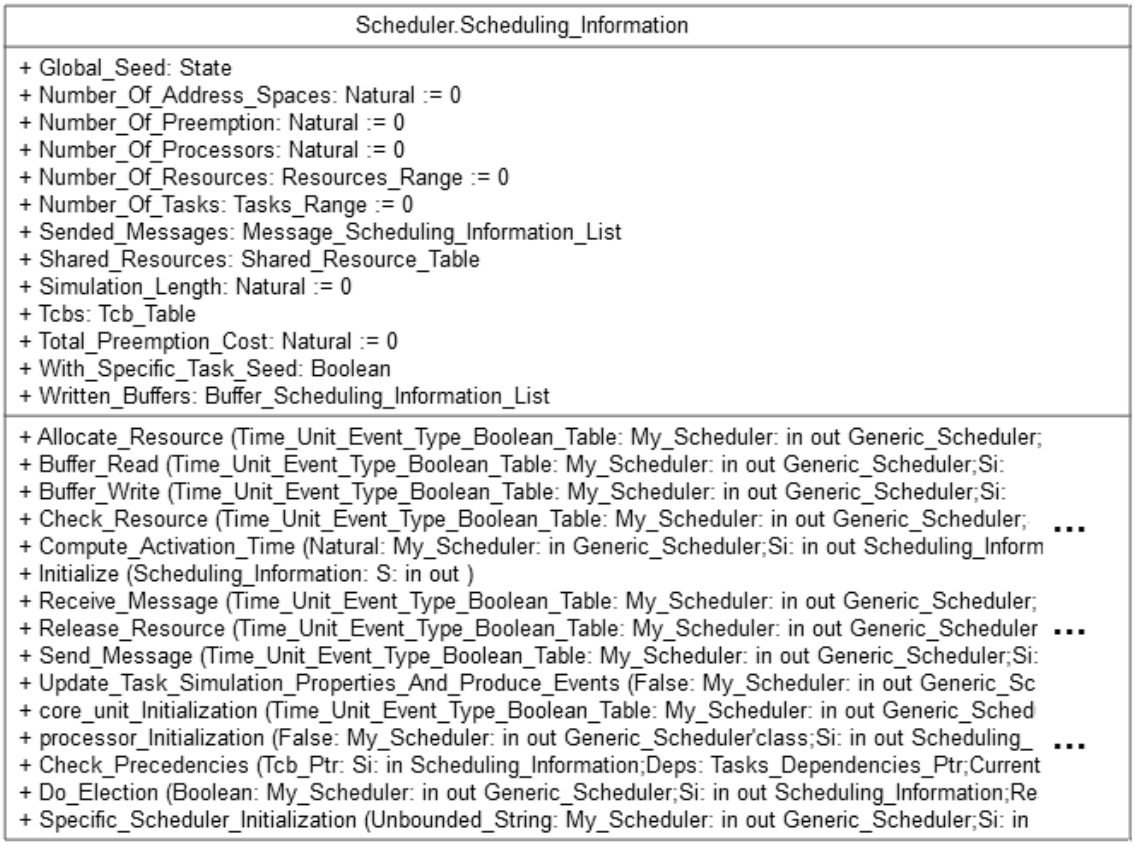

FIGURE 13 – Structure de données `Scheduling_Informations`

Nous pouvons voir qu'au sein de cette structure de données, en plus des informations propres à la simulation tel que le nombre de préemptions, ou le nombre de ressources partagées nous avons une table `Tcbs`. Cette table contient des informations essentielles pour chaque unité de temps de la simulation, et ce sur les différentes tâches. Ainsi nous avons pour chaque tâche ses différents états comme par exemple si est elle est active ou non, le reste de sa capacité, l'instance de la tâche etc. comme nous pouvons le voir sur la Figure 14.

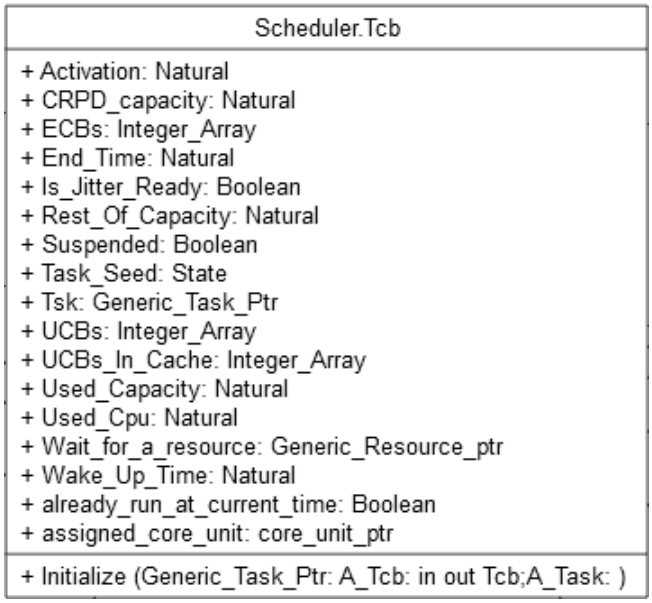

FIGURE 14 – Structure de données Tcb

Concernant le traitement des données de simulation, comme par exemple pour un affichage, celle-ci sont stockées au sein d'une table d'évènements. Un évènement est modélisés par la structure de données `Time_Unit_Event` qui caractérise un évènement à une unité de temps spécifique. Ainsi à chaque unité de temps il existe une ou plusieurs instances de `Time_Unit_Event` décrivant un évènement spécifique tel une préemption comme nous pouvons le voir sur la Figure 15.

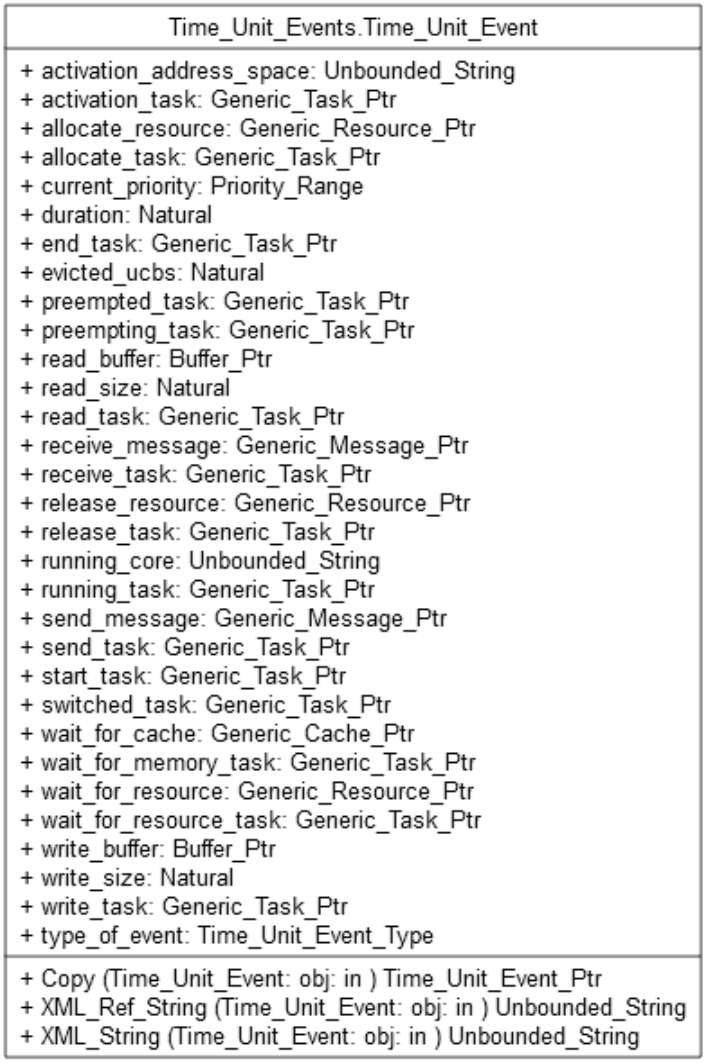

FIGURE 15 – Structure de données `Time_Unit_Event`

Maintenant que nous avons décrit les différentes information relatives aux données de simulation nous pouvons illustrer le code de la boucle de simulation dans l'annexe A, comme cité plus avant, avec des commentaires explicatifs afin que le lecteur puisse apprécier les modifications qui y seront effectuées par la suite.

Seront décrites dans les sous parties suivantes, les modifications qui seront apportées à la boucle de simulation ainsi que en moindre mesure à d'autres parties du logiciel. Nous aborderons l'implémentation de la modification de la boucle de si-

mulation, l'ajout d'une communication avec le logiciel AADLInspector ainsi que la modélisation et l'intégration du système d'heuristiques et de la logique interne des heuristiques. Enfin nous terminerons par une description des tests effectués afin de valider la solution, ses apports mais aussi ses limitations. Au sein de chacunes de ses sous parties suivantes seront abordés les problèmes rencontrés, si présents, ainsi que la manière de les résoudre. Nous pouvons dorénavant aborder la partie concernant la modification de la boucle de simulation de Cheddar, après conclusionde cette partie.

6.1.3 Conclusion

Cette analyse d'architecture nous a permis d'identifier différentes informations relatives au logiciel et particulièrement à sa partie simulation. En effet nous avons pu aborder la structure du logiciel et de quelle manière s'articulent les modules. Nous avons par la suite pu voir de quelle manière était appelé le module de simulation et comment celui-ci s'inscrivait dans le reste de l'outil. De plus, nous avons pu aborder le fonctionnement de la simulation dans Cheddar et ses limitations mais aussi les informations sur les données de simulations et comment celles-ci sont utilisées et stockées.

Cette étape nous a donc permis de prendre connaissance de l'outil et de son architecture mais surtout de connaître le fonctionnement interne de la simulation d'ordonnancement afin de pouvoir y apporter des modifications par la suite.

6.2 Implémentation des fonctionnalités

6.2.1 Modification de la boucle de simulation

En premier lieu, nous allons rappeler la problématique liée à la modification du fonctionnement de la boucle de simulation, sa portée et dans quel contexte cette modification est effectuée. En second lieu nous décrirons la méthode utilisée afin de répondre à l'objectif, et enfin nous terminerons par une illustration de la solution.

L'objectif final est l'intégration et le test d'au moins une heuristique au sein de Cheddar afin de vérifier la validité de cette solution et son efficacité. Cependant, en parallèle de ce but différents objectifs sont à réaliser en amont. En effet, comme nous l'avons vu dans la définition des problématique du projet, un des objectifs est de faire communiquer Cheddar et AADLInspector en établissant une communication

via Sockets. Préalablement à cet objectif est le fait de modifier la boucle de simulation afin d'effectuer pour chaque unité de temps les calculs d'ordonnancement lui étant spécifique et ainsi avec l'objectif précédent de faire communiquer les évènements à AADLInspector dès leur calcul effectué, et non pas à la fin de la simulation. Le comportement de la simulation de Cheddar sera donc basé sur le comportement de Marzhin en terme de communication. Cheddar et Marzhin pourrons donc être interchangés au sein d'AADLInspector sans manipulation complexe [28].

6.2.1.1 Mise en place de la communication

La mise en place de la communication est soumise à des contraintes. Ces contraintes sont propres au fonctionnement de la communication déjà mise en place entre AADLInspector et Marhzin. En effet, nous avons vu dans la partie problématique d'intégration, la manière dont AADLInspector utilisait les outils Marzhin et Cheddar et dans la partie définition des objectifs du stage, le fait de pouvoir interchanger les deux logiciels. Nous choisissons pour la communication entre AADLInspector et Cheddar de produire les mêmes évènements que Marzhin et doncd'être homogène dans la politique de communication des évènements. Ainsi, nous pouvons donner les messages de génération d'évènements existant pour Cheddar et Marzhin :

Cheddar	Marzhin
RUNNING_TASK	THREAD_STATE_RUNNING
TASK_ACTIVATION	THREAD_STATE_READY
PREEMPTION	THREAD_STATE_SUSPENDED
START_OF_TAST_CAPACITY	THREAD_DISPATCH
END_OF_TASK_CAPACITY	THREAD_STATE_SUSPENDED
WAIT_FOR_RESOURCE	THREAD_STATE_AWAITING_RESOURCE
ALLOCATE_RESOURCE	THREAD_GET_RESOURCE
RELEASE_RESOURCE	THREAD_RELEASE_RESOURCE

Table 1 – Comparatif des évènements Cheddar et Marzhin équivalents

28. e.g seul l'appel interne à AADLInspector en ligne de commande d'un des deux simulateurs sera à effectuer.

Actuellement, trois sockets sont définies entre AADLInspector et Marhzin, une socket permettant d'envoyer des commandes au simulateur Marzhin, une socket permettant de recevoir les messages de Marhzin et une dernière socket permettant de vérifier la bonne réception des commandes.

Nous allons donc reprendre le même schéma de communication à trois socket et établir une génération des messages de simulation identique à celle du simulateur Marzhin. Concernant la partie génération des messages, nous définissons un package `Marzhin_Utils` dans lequel seront décrites les différentes procédures, fonctions et variables utiles au traitements des données Cheddar afin de générer des messages événementiels du même type que Marzhin. Aucun problème majeur n'a été rencontré durant cette étape. En effet, celle-ci consiste principalement à manipuler des chaînes de caractères, ainsi il n'est pas nécessaire d'expliciter plus qu'avec un résumé cette fonctionnalité. Le fonctionnement de cette génération est donc le suivant :

- Cheddar génère une chaîne de caractères correspondant à l'évènement considéré après calcul, cet évènement est ensuite traité afin de modifier le type de l'évènement au type d'évènement Marzhin (procédure `Set_To_Marzhin_Event`). De plus, les différentes informations [29] nécessaires supplémentaires liées au type d'évènement sont obtenues puis l'ensemble des informations nécessaires sont structurées au format de sortie de Marhzin -

Maintenant que nous avons une sortie des évènements identique au format de sortie des évènements Marzhin nous allons pouvoir mettre en place la communication entre AADLInspector et Cheddar. Dans cette optique, différentes librairies proposant des fonctions pour la communication par sockets ont été étudiées. Ainsi, la communication sera développée en se basant sur la libraire GNAT.Sockets livrée de façon standard avec le compilateur *GNAT*. Des premiers tests ont été effectués hors du contexte de Cheddar afin d'établir une communication entre deux programmes triviaux en Ada afin d'avoir un mécanisme de communication valide hors du contexte.

La communication étant validée entre ces deux programmes certaines problématiques restent en suspend. En effet, ici il n'est nul question de communication entre deux machines différentes, ni de communication dans un format spécifique. Nous pouvons dorénavant ajouter ce mécanisme de communication mais de façon adaptée à Cheddar. Un des points important dans cette intégration est le placement de la communication de manière efficace. En effet, un effort à été fait dans la conception de

29. e.g id tâche, id ressource, etc.

la solution globale afin d'optimiser au mieux le placement des différentes fonctionnalités et eviter au maximum les entrelacements et dépendances. Ainsi nous activerons la connection au début de la procédure `Build_Multiprocessor_Scheduling` sur les trois sockets.

Enfin, au sein de la boucle de simulation nous transmetterons les données de simulation au simulateur Marzhin, Listing 3 ci-dessous.

```
 if Socket_Mode_Activated then
     AADLInspector_Data_Communication(Sys, Result, J, Current_Time(J),
↪  Last_Time_Mod, SpeedFactor, Speed, Slice_Size, Exit_Simulation);
 end if;
```

Listing 3: Communication des données de simulation

Cette communication se base sur un package que nous avons développé se nommant `Sockets_Overlay` et ajoutant une surcouche à la communication par sockets proposée par GNAT.Sockets. Cette surcouche ajoutée par ce package nous permet de définir des procédures simplifiant notre modèle de communication et ainsi augmenter la maintenance et la compréhension du code produit. Nous pouvons voir dans le Listing 3 un appel à la procédure `AADLInspector_Data_Communication` que nous avons défini, permettant d'effectuer les traitements inhérents à la communication des évènements entre AADLInspector et Cheddar.

Cependant, lors du premier test d'envoi de messages, une problématique est apparue. En effet, une partie du message d'évènement envoyé à AADLInspector n'est pas reçu. Ce sont 8 caractères qui sont perdus, après une première recherche la cause n'est pas identifiée au sein du code. Afin de poursuivre les tests, le message est modifié à la reception du coté d'AADLInspector afin de prendre en compte ce décalage de 8 caractères non identifiés.

Maintenant que l'envoi d'évènement est validé, la mise en place des sockets de commande et d'acknoledgment sont effectuées. AADLInspector peut effectuer différentes actions sur le simulateur Marzhin à partir de l'interface graphique comme nous pouvons le voir en haut à gauche de la Figure 16 ci-dessous.

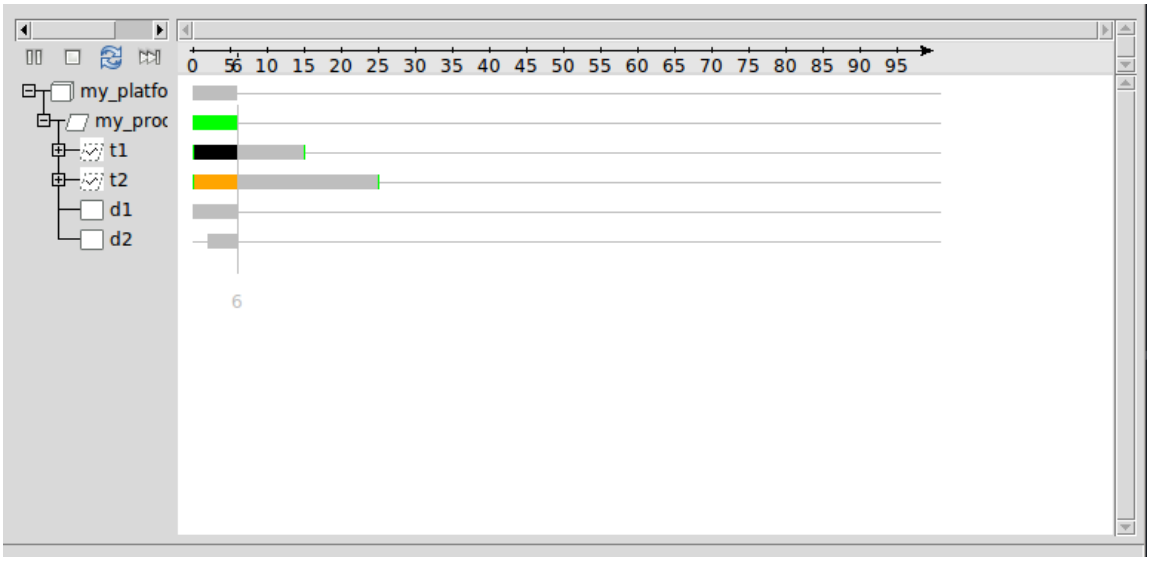

FIGURE 16 – Chronogramme produit par AADLInspector et boutons de contrôle

Nous allons donc mettre en place la reception des commandes ainsi que le comportement associé. Les différentes commandes et comportement associés sont définis au sein de l'Annexe B.

Nous pouvons donc voir que les comportement en reception de ces différentes commandes induisent implicitement une dépendance assez forte entre ce code, et le code de la simulation en elle-même. Afin de confirmer le fonctionnement, les tests ont d'abord été effectués en local avec un autre programme Ada permettant d'envoyer les commandes et de recevoir les messages à la place d'AADLInspector. Une fois le comportement de l'envoi de commande validé vis à vis du comportement de la simulation attendu, les tests ont été effectués avec AADLInspector.

Un problème majeur est apparu directement au lancement de la simulation. En effet, Cheddar reste en attente malgré les sockets connectées et l'envoi du message

de commande « play » depuis AADLInspector. Le code effectué a été entièrement revu, déboguer et aucune information n'a pu émerger de cette analyse permettant de corriger le problème. Une lecture a été faite du code et de la documentation de GNAT.Socket et aucune information n'a pu être trouvée. Plusieurs recherches sur cette problématique ont été effectués sur Internet et aucun résultat n'a été trouvé. A donc été fait une analyse des paquets réseaux entre Cheddar et Marzhin afin de vérifier que la commande est bien envoyée et que son format est correct.

Néanmoins, en analysant les messages envoyés de Cheddar à Marzhin nous avons pu observer en plus de la chaîne de l'évènement, 8 octets non voulus. Cela reviens à la problématique non résolue précédemment dans l'envoi des messages. Une surcouche doit être présente au sein de GNAT.Socket qui n'est pas documentée et qui doit ajouter des informations sur le message envoyé mais doit probablement attendre ses informations aussi en réception. Ce qui expliquerait le blocage. Après une recherche plus ciblée sur le terme de surcouche réseau, j'ai trouvé un extrait [30] de livre qui aborde cette problématique. En effet, GNAT.Socket ajoute une surcouche qui ajoute un protocole spécifique au dessus de la pile TCP, lors de l'envoi d'une chaîne de caractères via la fonction `String'Output(s)`, la chaîne est transmise sous la forme :

- un entier avec la valeur 1 sur 4 octets en binaire suivi de,
- un entier sur 4 octets en binaire qui représente la longueur de la chaîne,
- et enfin les caractères de la chaîne *s*

La fonction `String'Input` n'accepte elle aussi que ce format.

Après correction, la réception de commande, l'envoi de message et d'acknoledgment ont été validés. À ce stade du projet, la boucle de simulation a été modifiée afin de générer des messages concernant les évènements à chaque unités de temps et une communication par sockets a été mise en place entre Cheddar et AADLInspector permettant de visualiser dans AADLInspector un ordonnancement actuellement en cours de calcul au sein de Cheddar mais aussi de contrôler le déroulement de la simulation au sein de Cheddar en modifiant la vitesse d'envoi des messages, en mettant en pause la simulation, en l'arrêtant ou encore en modifiant le temps final de simulation. Le simulateur Marzhin et le simulateur Cheddar peuvent dorénavant être

30. [11] Francis Cottet et Emmanuel Grolleau, *Systèmes temps réel embarqués - 2e éd : Spécification, conception, implémentation et validation temporelle*, Dunod, p.435.

interchangés au sein d'AADLInspector de façon transparente. Nous avons dorénavant le support de base afin de développer les heuristiques définies et ainsi visualiser en temps-réel les gain effectués lors des sauts temporels en utilisant Cheddar via AADLInspector.

6.2.2 Implémentation du système d'heuristiques

Dans cette section, nous traitons de l'implémentation du système d'heuristique et de sa conception de sorte à faciliter l'intégration de nouvelles heuristiques, la modification d'heuristiques existantes ou d'autres actions sur tout ou partie des heuristiques.

Afin de concevoir une architecture répondant à nos besoin il faut penser à la manière dont les heuristiques pourraient être utilisées. Ainsi, nous pourrions utiliser une ou plusieurs heuristiques. Il serait même possible d'avoir une liste d'heuristiques et cette liste générée automatiquement en fonction de l'ordonnanceur et du système considéré afin d'avoir les heuristiques compatibles sans autre activation de la part de l'utilisateur, hormis l'activation globale des optimisation, si non activée par défaut. Cependant ayant défini trois heuristiques pour l'implémentation, la génération automatique d'une liste compatible n'est pas nécessaire mais l'architecture du système d'heuristiques est effectuée de telle sorte à rendre cela possible.

Ainsi, si nous pouvons définir les différentes exigences :

- *Le système doit avoir des heuristiques*
- *Le système doit contenir une ou plusieurs heuristiques.*
- *Le système doit contenir une liste d'heuristiques.*
- *Les heuristiques doivent avoir des comportements différents.*
- *Les heuristiques doivent avoir des conditions d'exécution différentes.*
- *Une heuristique possède des données qui lui sont propres.*
- *Une heuristique peut posséder des méthodes qui lui sont propres.*
- *Une heuristique doit implémenter des méthodes spécifiques.*

À partir de ces exigences nous pouvons définir le diagramme UML suivant :

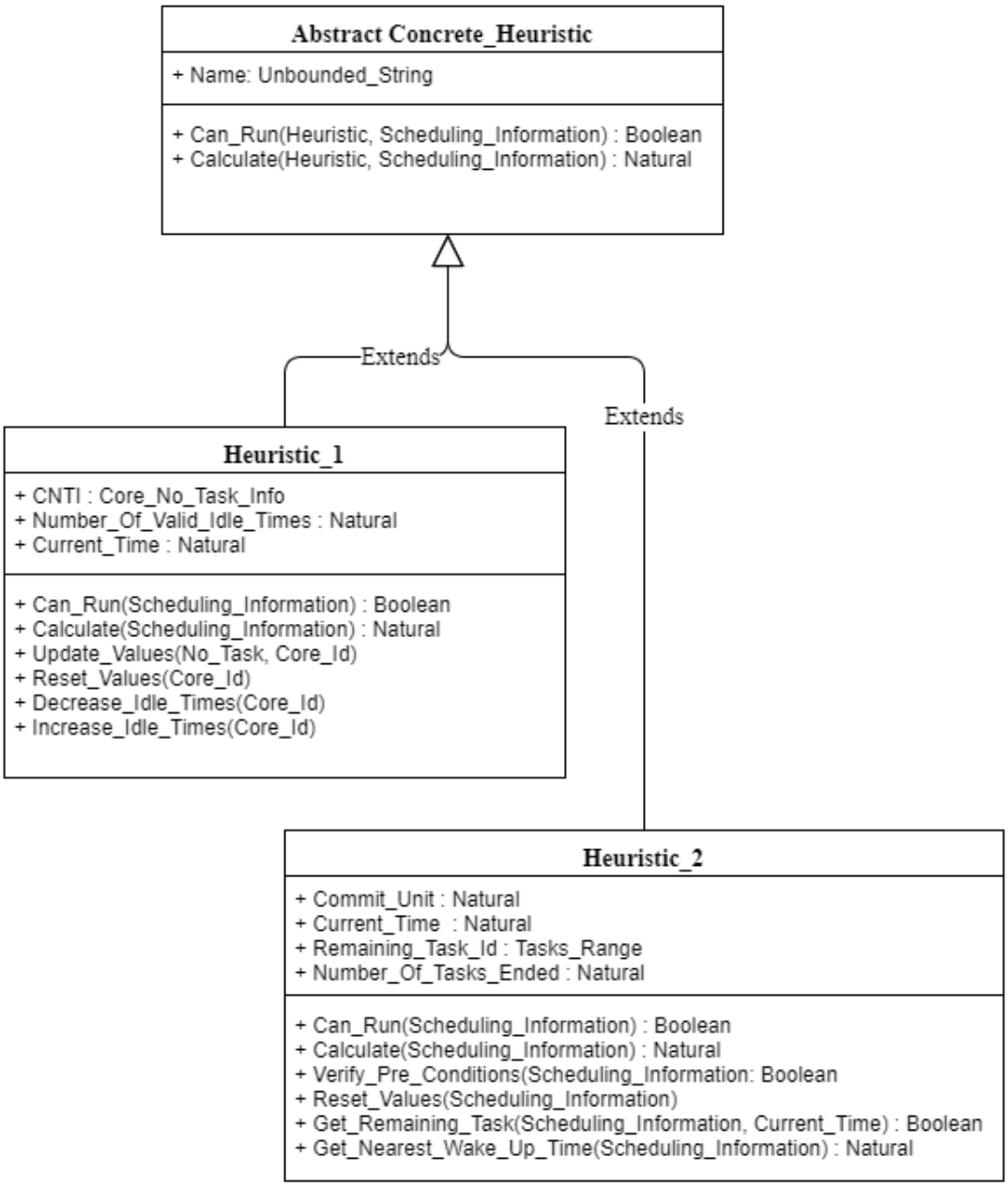

FIGURE 17 – Diagramme UML du système d'heuristiques

Ainsi avec cette architecture, nous pouvons profiter du polymorphisme et ainsi avoir des fonction uniques permettant de passer une heuristique de type X ou Y sans spécification. De plus, nous pouvons avoir une liste d'heuristiques composées d'heuristiques différentes. Dans ce schéma UML, seul deux heuristiques sont définies, mais d'autres peuvent être ajoutées et hériter de `Concrete_Heuristic`. Nous définissons donc dans le Listing 4 une interface `Heuristic` possédant deux fonctions abstraites : `Can_Run` et `Calculate` qui sont présentes dans toute heuristique.

```
type Heuristic is interface;

-- Abstract functions and procedures:
function Can_Run(This : in out Heuristic;
                 Si   : in     Scheduling_Information)
return Boolean is abstract;

function Calculate(This : in out Heuristic;
                   Si   : in     Scheduling_Information)
return Natural is abstract;
```

Listing 4: Définition de l'interface `Heuristic`.

Afin de permettre le polymorphisme dans la version Ada utilisée (Ada 2005) un type tampon est utilisé en tant que type Abstrait sous le nom `Concrete_Heuristics` et possèdant un nom et un pointeur est définit sur ce type comme nous pouvons le voir dans le Listing 5 suivant :

```
-- Abstract type of an Heuristic:
type Concrete_Heuristics is abstract new Heuristics.Heuristic
  with record
    Name : Unbounded_String;
    -- *
  end record;

type Heuristic_Ptr is access Concrete_Heuristics'Class;
```

Listing 5: Définition du type `Concrete_Heuristics`.

Les types peuvent ensuite être définis comme dérivés du type `Concrete_Heuristics` et possédants leurs propres attributs sur lesquels nous reviendrons par la suite.

```
type Heuristic_1 is new Concrete_Heuristics
  with record
    CNTI : Core_No_Task_Info := (others => 0);
    Number_Of_Valid_Idle_Times : Natural := 0;
    Current_Time : Natural := 0;
  end record;

type Heuristic_2 is new Concrete_Heuristics
  with record
    Commit_Unit  : Natural := 0;
    Current_Time : Natural := 0;
    Remaining_Task_Id : Tasks_Range := 0;
    Number_Of_Tasks_Ended : Natural := 0;
  end record;
```

Listing 6: Définition des types d'heuristiques.

Les méthodes surchargés [31] propres à l'ensemble des heuristiques - `Can_Run` et `Calculate` - sont ensuites définies (Listing 7) et enfin les méthodes propres à chaque heuristique sont définies (Listing 8). Enfin une liste d'heuristiques est définie (Listing 9).

31. La surcharge permet de choisir entre différentes versions d'une même fonction ou méthode selon le nombre et le type des arguments fournis

```
-- Heuristics functions and procedures:
function Can_Run (This : in out Heuristic_1;
                  Si   : in     Scheduling_Information)
return Boolean;

function Can_Run (This : in out Heuristic_2;
                  Si   : in     Scheduling_Information)
return Boolean;

function Calculate (This : in out Heuristic_1;
                    Si   : in Scheduling_Information)
return Natural;

function Calculate (This : in out Heuristic_2;
                    Si   : in     Scheduling_Information)
return Natural;
```

Listing 7: Définition des méthodes surchargées.

```
-- Proper to Heuristic_1::Idle_Heuristic:
procedure Update_Values (This    : in out Heuristic_1;
                         No_Task : in     Boolean;
                         Core_Id : in     Natural);

procedure Reset_Values (This    : in out Heuristic_1;
                        Core_Id : in     Natural);
--
procedure Decrease_Idle_Times (This    : in out Heuristic_1;
                               Core_Id : in     Natural);

procedure Increase_Idle_Times (This    : in out Heuristic_1;
                               Core_Id : in     Natural);

-- Proper to Heuristic_2::Last_Job_Remaining_Heuristic:
function  Verify_Pre_Conditions (This : in out Heuristic_2;
                                 Si : in Scheduling_Information)
return Boolean;

procedure Reset_Values (This : in out Heuristic_2;
                        Si   : in Scheduling_Information);
--
function Get_Remaining_Task (This         : in out Heuristic_2;
                             Si           : in     Scheduling_Information;
                             Current_Time : in     Natural)
return Boolean;

function Get_Nearest_Wake_Up_Time (This : in out Heuristic_2;
                                   Si   : in     Scheduling_Information)
return Natural;
```

Listing 8: Définition des méthodes propres aux heuristiques.

```
-- Heuristic list:
package Heuristic_Lists is new Ada.Containers.Doubly_Linked_Lists
↪  (Heuristic_Ptr);
use Heuristic_Lists;
Heuristic_List : List;
```

Listing 9: Définition d'une liste d'heuristiques.

Les méthodes `Can_Run` et `Calculate` de l'interface, doivent être implémentées pour chaque heuristique. En fonction de l'heuristique considérée, un comportement différent est attribué à ces deux méthodes. Le rôle de `Can_Run` est de mettre en place un test sur les conditions d'exécution de l'heuristique. Ainsi, si cette méthode retourne la valeur *vrai* lorsque les conditions d'exécution sont valides et ainsi un appel à la méthode `Calculate` qui elle effectue le calcul de l'heuristique est effectué.

Maintenant que nous avons décrit la manière dont est implémenté le système d'heuristiques afin de garantir une maintenance et une évolution simple de celui-ci, nous allons aborder l'implémentation des heuristiques en elles-même au sein de Cheddar.

6.2.3 Implémentation des heuristiques

Dans cette section nous allons aborder la manière dont ont été implémentées les différentes heuristiques au sein de Cheddar conformément aux considérations des aspects théoriques que nous avons abordés dans la partie y étant dédiée.

Nous allons implémenter au sein de Cheddar deux heuristiques. La première est l'heuristique basée sur les temps creux et la seconde sur le dernier travail actif. Dans une considération de gestion du temps en rapport avec les autres travaux à effectuer durant le stage et le fait d'ajouter une heuristique supplémentaire ou non n'étant pas une exigence nous n'implémenterons pas l'heuristique basée sur les systèmes à ordonnancement non-préemptif. En effet, le temps est limité et les tâches à effectuer sont multiples et le but est seulement d'étudier la possibilité d'utiliser les heuristique à des fin d'optimisation et non de définir et d'implémenter toutes les heuristiques définissables.

Un des objectifs implicites majeur est d'implémenter la solution au sein de Cheddar en minimisant au maximum la modification du code existant. En effet, il faut éviter au maximum de modifier les méthodes au sein du module de simulation sous peine de fausser les calculs d'ordonnancement.

La solution doit donc se visualiser comme une sonde se branchant à une méthode spécifique qui est `Build_Multiprocessor_Scheduling` et ainsi la quasi-totalité du code fonctionnel est présent dans des fichiers spécifiques hors de tout fichier déjà existant. De plus, un mécanisme de contrôle des heuristiques a été ajouté, permettant simplement d'activer ou non les heuristiques par ligne de commande au lancement de Cheddar. L'option à ajouter au lancement du kernel Cheddar est l'option `> -optimize`.

6.2.3.1 Temps creux

Ainsi, nous allons décrire la première heuristique basée sur les temps creux. Un des points important à considérer est la manière de détecter les évènements qui nous intéressent. Ici, nous cherchons premièrement à détecter lorsque aucune tâche n'est en cours d'exécution. Aucun évènement n'est produit pour décrire cette situation. Nous devrons l'obtenir autrement. Une des manière est de détecter lorsque une tâche termine sa capacité et d'avoir un compteur permettant d'obtenir une indication sur le nombre de tâches actives. Nous pouvons donc définir la table `CNTI` (Core No Task Information) de longueur égale au nombre de coeurs du système et contenant pour chaque coeur le nombre de tâches ayant terminé l'exécution de leur travail courant.

Ainsi, notre première condition sera que sur l'ensemble des coeurs, aucune tâche ne soit active. Il faut cependant ajouter un critère qui est propre à Cheddar qui est le fait que l'évènement est généré une unité de temps après au niveau du traitement. Ainsi on attendra au minimum 2 unités de temps durant lesquelles aucune activité est présente afin d'éviter de modifier le comportement interne de Cheddar.

On obtient donc trois procédure de contrôle du nombre de temps creux étant monitorés : `Reset_Values` qui permet de réinitialiser les différents attributs de l'instance de l'heuristique, `Decrease_Idle_Times` qui permet de diminuer le nombre de temps creux monitorés et `Increase_Idle_Times` qui permet d'effectuer l'opération inverse (Listing 10).

```
 -------------------------
 -- Heuristic_1::Update --
 -------------------------

 procedure Update_Values (This : in out Heuristic_1; No_Task : in Boolean;
↪  Core_Id : in Natural) is
 begin
   if No_Task then
     Increase_Idle_Times (This, Core_Id);
   else
     Decrease_Idle_Times (This, Core_Id);
   end if;
 end Update_Values;

 -------------------------------
 -- Heuristic_1::Reset_Values --
 -------------------------------

 procedure Reset_Values (This : in out Heuristic_1; Core_Id : in Natural) is
 begin
   This.CNTI(Natural(core_id)) := 0;
   This.Number_Of_Valid_Idle_Times := 0;
 end Reset_Values;

 ---------------------------------------
 -- Heuristic_1::Decrease_Idle_Times --
 ---------------------------------------

 procedure Decrease_Idle_Times (This : in out Heuristic_1; Core_Id : in
↪  Natural) is
 begin
   This.CNTI(Natural(core_id)) := 0;
   This.Number_Of_Valid_Idle_Times :=
     (if This.Number_Of_Valid_Idle_Times = 0 then 0 else
↪  This.Number_Of_Valid_Idle_Times - 1);
 end Decrease_Idle_Times;

 ---------------------------------------
 -- Heuristic_1::Increase_Idle_Times --
 ---------------------------------------

 procedure Increase_Idle_Times (This : in out Heuristic_1; Core_Id : in
↪  Natural) is
 begin
   This.CNTI(Natural(core_id)) := This.CNTI(Natural(core_id)) + 1;
   if This.CNTI(Natural(core_id)) = 2 then
     This.Number_Of_Valid_Idle_Times := This.Number_Of_Valid_Idle_Times + 1;
   end if;
 end Increase_Idle_Times;
```


Listing 10: Code des méthodes de l'heuristique 1 (temps creux).

Concernant la méthode `Calculate` de cette première heuristique, nous appliquons l'algorithme 1 que nous avons défini au sein de la définition de notre approche théorique :

```
 ----------------------------
 -- Heuristic_1::Calculate --
 ----------------------------

 function Calculate (This : in out Heuristic_1; Si : in
↪  Scheduling_Information) return Natural is
     Nearest_Wake_Up_Time : Natural := Natural'Last;
 begin
   Put_Debug("__INFO__ :: Number_Of_Valid_Idle_Times -> TRUE");
   for num_task in 1 .. Si.Number_Of_Tasks - 1 loop
     if Si.Tcbs(num_task).Wake_Up_Time < Nearest_Wake_Up_Time then
       Nearest_Wake_Up_Time := Si.Tcbs(num_task).Wake_Up_Time;
     end if;
   end loop;
   Put_Debug("__INFO__ :: Nearest_Wake_Up_Time -> " &
↪  Nearest_Wake_Up_Time'Img);
   Return Nearest_Wake_Up_Time;
 end Calculate;
```

Listing 11: Code de la fonction `Calculate` de l'heuristique 1 (temps creux).

Enfin, pour terminer l'implémentation de notre première heuristique, son appel doit être effectué au sein de la méthode `Build_Multiprocessor_Scheduling`, comme nous pouvons le voir dans le Listing 12.

```
-------------------------------------------------------------
-- Computing of the idle times based heuristic
-------------------------------------------------------------
First_Heuristic:
declare
  -- local variable used only for first heuristic calculations.
  Jump_Time : Natural;
begin
Update_Values (Heuristic_1(Idle_Heuristic.all), No_Task, Natural(core_id));

if Heuristic_1(Idle_Heuristic.all).Number_Of_Valid_Idle_Times =
Natural(table_of_core_scheduler(j).nb_entries) then
  Jump_Time := Calculate_Heuristic (Idle_Heuristic.all, Si);

  if Jump_Time > Current_Time (J) + 2 then
    Heuristic_1(Idle_Heuristic.all).Current_Time := Current_Time(J);
    if table_of_core_scheduler(j).entries (core_id).scheduler.all in
    Hierarchical_*  then
      Increase_Used_CPU(Hierarchical_*(
      table_of_core_scheduler(j).entries (core_id).scheduler.all),
      (Jump_Time-Current_Time(J)-1));
      Jump_Time := Jump_Time - 1;
    end if;

    Current_Time (J)  := Jump_Time;
    Last_Current_Time := Current_Time (J);
    Reset_Values (Heuristic_1(Idle_Heuristic.all), Natural(core_id));
    goto BACK_TO_THE_FUTURE;
  end if;
end if;
end First_Heuristic;
```

Listing 12: Appel de l'heuristique 1 (temps creux) dans `Build_Multiprocessor_Scheduling`.

Maintenant que nous avons présenté l'ensemble des méthodes propres à l'heuristique basée sur l'élimination des temps creux et son intégration au sein de Cheddar, nous allons aborder la seconde et dernière heuristique que nous intégrerons.

6.2.3.2 Dernier travail actif

Cette heuristique est comme nous l'avons vu, valable lorsque une seule tâche est actuellement en cours d'exécution. Les différentes contraintes et conditions de cette heuristiques ont été définies au sein de la partie lui étant dédiée au sein de la partie définition des heuristique pour tâche unique. Nous seront plus succins dans la présentation de cette heuristique sachant que la manière de l'intégrer est semblable.

L'extrait de code présenté dans le Listing 6 concerne les attributs des différentes heuristiques. Nous avons ici, pour la seconde heuristique des attributs qui lui sont propres et utiles au calcul de cette dernière. Ainsi, l'attribut `Commit_Unit` permet de monitorer l'instant à partir duquel l'évènement a été généré. Les autres attributs utiles à cette heuristiques parlent par leur noms.

La première méthode utile à cette heuristique est la vérification des conditions d'exécution de l'heuristique. Ces différentes vérifications permettent d'assurer que l'heuristique ne soit pas exécutée dans des conditions qui nous qualifions d'instables et pouvant perturber la validité de l'ordonnancement généré. Ainsi, la méthode présentée dans le Listing 7 page suivante, `Verify_Pre_Conditions`, permet de tester les différents cas d'instabilité potentielle.

```
   ----------------------------------------
   -- Heuristic_2::Verify_Pre_Conditions --
   ----------------------------------------

   function Verify_Pre_Conditions (
     This              : in out Heuristic_2;
     Si                : in     Scheduling_Information)
   return Boolean is
   begin
     -- Test the number of remaining jobs:
     if not Get_Remaining_Task(This, Si, This.Current_Time) then
       return False;
     end if;

     if (Si.Tcbs(This.Remaining_Task_Id).Rest_Of_Capacity >
↪     Si.Tcbs(This.Remaining_Task_Id).Tsk.capacity - 2) then
       This.Commit_Unit := Si.Tcbs(This.Remaining_Task_Id).Tsk.capacity - 2;
     end if;

     -- Test the capacity of the remaining job:
     if Si.Tcbs(This.Remaining_Task_Id).Tsk.capacity <= 2 then
       return False;
     end if;

     [...]
```

Listing 13: Code de la fonction de vérification des conditions de l'heuristique 2 (tâche unique) [Part 1.].

```
    [...]

    -- Test the starting event commit:
    if Si.Tcbs(This.Remaining_Task_Id).Wake_Up_Time > This.Current_Time
    or Si.Tcbs(This.Remaining_Task_Id).Rest_Of_Capacity >= This.Commit_Unit
    then
      return False;
    end if;

    -- Test the nearest wake up time:
    if (Get_Nearest_Wake_Up_Time(This, Si) - This.Current_Time) <= 2
    then
      return False;
    end if;

    -- Test the jump time:
    if ((This.Current_Time +
↪    Si.Tcbs(This.Remaining_Task_Id).Rest_Of_Capacity) - This.Current_Time) <=
↪    2 then
      return False;
    end if;

    return True;
  end Verify_Pre_Conditions;
```

Listing 14: Code de la fonction de vérification des conditions de l'heuristique 2 (tâche unique) [Part 2.].

Les trois autres méthodes inhérentes à cette heuristiques sont `Get_Remaining_Task, Get_Nearest_Wake_Up_Time` et `Reset_Values`. Celles-ci sont des méthodes utilitaires permettant d'obtenir des valeurs utiles au calcul de l'heuristique et de réinitialiser les valeurs des attributs de l'heuristique. Le code de ces méthodes est disponible en Annexe C.

L'appel à l'heuristique est défini au sein de `Build_Multiprocessor_Scheduling` de la manière suivante :

```
-------------------------------------------------------------------
-- Computing of the last job running based heuristic
-------------------------------------------------------------------
Heuristic_2(Last_Job_Heuristic.all).Current_Time := Current_Time(J);
if Verify_Pre_Conditions(Heuristic_2(Last_Job_Heuristic.all), Si) then
  Current_Time(J)   := Calculate_Heuristic(Last_Job_Heuristic.all, Si);
  Last_Current_Time := Current_Time (J);
  Reset_Values(Heuristic_2(Last_Job_Heuristic.all), Si);
  goto BACK_TO_THE_FUTURE;
end if;
```

Listing 15: Code de l'appel à l'heuristique 2 dans `Build_Multiprocessor_Scheduling`.

La boucle de simulation de la méthode `Build_Multiprocessor_Scheduling` modifiée est disponible en Annexe D.

6.3 Validation de l'implémentation

Les heuristiques considérés pour implémentations étant intégrées à notre cas d'étude, une validation de l'implémentation est nécessaire afin de confirmer le comportement de l'ordonnanceur. En effet, les modifications apportées sont au coeur du moteur de la simulation et interfèrent avec l'ordonnancement lui-même. Il faut donc s'assurer que malgré les optimisations apportés par les heuristiques, aucune perturbation de la validité de l'ordonnancement n'est générée. Pour cela, une liste de modèles d'ordonnancement intégrés au projet Cheddar ont été utilisés afin de comparer les résultats de simulation sur différentes architectures, différents algorithmes d'ordonnancement et différents modèles de tâches.

Ces différents tests on permis au cours de l'intégration de corriger certains problèmes. Comme par exemple la prise en compte des algorithmes d'ordonnancement hiérarchisés qui lorsque l'optimisation sur les temps creux effectué doit modifier le nombre d'unités processeur utilisées sans quoi l'ordonnancement ne serait plus correct par la suite, comme nous pouvons le voir à l'appel de la procédure `Increase_Used_CPU` dans le Listing 12.

Des problèmes ont aussi été détectés au sein des cas de tests offerts par Cheddar, car en effet une partie de ceux-ci n'étaient pas valides dans l'ordonnancement qui devait être obtenu. Le problème a été remonté et est depuis corrigé. Après correction de ce problème l'ensemble des cas de tests pour Cheddar ont été effectués et intégralement validés. Le fichier Excel décrivant les différents cas et résultats obtenu peut-être trouvé en Annexe E.

6.4 Cas pratique Marzhin

L'idée principale à l'origine de la conception du logiciel Marzhin a été d'appliquer les concepts multi-agents à la simulation des élémments d'architecture AADL générés par les autres outils de modélisation de l'entreprise Ellidiss Technologies. L'objectif principal est de réaliser des simulations d'ordonnancement en se basant sur des modèles AADL. Ainsi, chaque élément d'architecture possède son propre comportement puis ces différents éléments d'architecture sont exécutés dans la simulation afin de faire émerger un comportement global représentant le système en exécution. L'intérêt est ici de pouvoir décrire de façon isolée le comportement de chaque élément, ce qui s'oppose à une simulation centralisée comme Cheddar qui doit prendre en compte les propriétés de chaque élément dans un seul comportement.

Les détails de l'intégration de la solution ne seront pas décrit précisément étant donné que la manière de l'intégrer au sein du simulateur Marzhin est fortement semblable. Cependant, l'Annexe F présente les différentes modifications apportés au simulateur afin d'appliquer le système d'heuristiques. Dans le cas présent, par manque de temps, seule la première heuristique basée sur les temps creux à pu être implémentée au sein de ce simulateur.

La méthode est restée la même telle que présentée avec le concept de monitoring des tâches terminant leur travail comme nous l'avons vu dans la partie 6.2.3.1. L'heuristique a ensuite été testée sur différents modèles intégrés au simulateur Marzhin. L'ensemble de ces tests ont été validés. Cependant, d'autres tests communs à Cheddar et Marzhin seraient les bienvenus afin de pouvoir comparer leur modèle d'exécution en rapport avec cette heuristique.

Ce mode d'optimisation par simulation hybrid-driven basé sur des heuristiques est donc applicable à des simulateurs basés sur d'autres modes de fonctionnement interne ainsi que de représentation des éléments à simuler.

7 Conclusion

Différents projets ont tentés d'apporter une optimisation des simulations d'ordonnancement en se basant sur différentes techniques et modèles de simulation tel que la simulation event-driven. Cependant, ces projets sont en règle générale spécifiques à certaines applications et n'ont donc pas un large panel d'algorithmes disponibles, ni un large modèle de composants architecturaux définissables pour le système. Le logiciel Cheddar est un outil d'analyse et de simulation d'ordonnancement offrant de nombreux algorithmes d'ordonnancement et permet aussi de définir ses propres politiques d'ordonnancement ainsi que l'architecture et le modèle de tâche souhaité en plus des différentes fonctionnalités d'analyse d'ordonnançabilité. Ce simulateur est time-driven et une problématique d'optimisation de la simulation a émergée afin de l'orienter vers une simulation événementielle. Dans ce but, une approche hybride entre la simulation event-driven et la simulation time-driven a été choisie permettant de profiter des avantages de la simulation event-driven dans des cas précis. Cela évite de plus de modifier l'ensemble de l'architecture interne du logiciel Cheddar. Ainsi, cette approche est basée sur un système d'heuristiques qui permettent de définir des intervalles durant lesquelles un saut temporel peut-être effectué. Initialement prévu pour éviter les temps dit creux, où aucune tâche n'est exécutée, d'autres heuristiques ont été définies et donc notre système d'heuristiques étendu à d'autres intervalles de simulation. Cette approche a ensuite été développée et intégrée en langage Ada au sein de Cheddar qui nous aura servi de cas d'étude. Nous aurons ainsi répondu à notre problématique concernant l'optimisation en terme de temps d'exécution de la simulation et nombre d'évènement générés en se basant sur des heuristiques. De plus, nous aurons répondu à la problématique d'intégration permettant de transformer la simulation Cheddar en simulation pas à pas et de la contrôler via sockets. De plus, nous avons pu appliquer notre approche à un autre logiciel de simulation d'ordonnancement, Marzhin, dont le fonctionnement interne est basé sur un système multi-agent et ainsi diffère du fonctionnement de Cheddar. L'approche à elle aussi été validée au sein de ce cas pratique et donc suppose que l'approche que nous avons définie peut-être intégrée de façon simple au sein d'autres simulateurs d'ordonnancement. Cependant, par manque de temps sur la durée du stage, il n'a pas été possible d'achever une étude de performances sur l'approche hybride définie au cours de ce stage. Cependant, tout semble indiquer de manière théorique que tant que l'heuristique reste la plus généraliste possible et donc simple dans ses conditions, elle pourra apporter un bénéfice non négligeable en terme d'optimisation en fonction des modèles considérés.

Acronymes

AADL Architecture Analysis and Design Language. 6, *Glossaire :* AADL

DFG Data Flow Graph. 14, *Glossaire :* DFG

ETM Execution Time Model. 16, *Glossaire :* ETM

GPL GNU General Public License. 40, *Glossaire :* GPL

HOOD Hierarchical Object Oriented Design. 6, *Glossaire :* HOOD

MAST Modeling and Analysis Suite for Real-Time Applications (MAST) . 15, *Glossaire :* MAST

TLM Transaction-Level Modeling. 14, *Glossaire :* TLM

WCET Worst Case Execution Time. 16, *Glossaire :* WCET

XML Extensible Markup Language. 8, *Glossaire :* XML

Glossaire

Ada Langage de programmation développé dans les années 1980 au sein du Département de la Défence des États-Unis. Le langage était à l'origine modélisé pour les systèmes embarqués et temps-réel. Au moment de la rédaction de ce rapport, la dernière mise à jour stable est Ada 2012. Les mises à jour précédentes du langage incluent Ada 83, Ada 95 et Ada 2005 . 9

DFG Un Data Flow Graph (DFG) est un graphe représentant les dépendances en terme de données entre différentes opérations . 6, 14

event-driven Une simulation event-driven implique et exécute des évènements à des unités de temps arbitraires. En effet, la simulation se dirige de manière à passer d'évènement en évènement et non pas de passer par chaque unité de temps. Ce comportement s'explique par le fait que ce soit les évènements qui implique la présence d'autres évènements futurs. Ainsi, dans ce type de simulation, aucune optimisation de l'intervalle de temps ne doit être effectuée. 3

GNAT GNAT est un compilateur Ada open-source supprtant les versions Ada 83, Ada 95, Ada 2005 et Ada 2012. 50

GPL La licence publique générale GNU, ou GNU General Public License est une licence qui fixe les conditions légales de distribution d'un logiciel libre du projet GNU. 40

HOOD Méthode de modélisation et de processus de développement créée par l'Agence Spaciale Européenne avec comme but initial d'effectuer de l'ingénierie dirigée par les modèles et en offrant une solution de modélisation pour le langage Ada utilisé dans ses projets de grande envergure. 6, 16

MAST Suite d'outils permettant d'effectuer de l'analyse d'ordonnancement pour systèmes temps-réels distribués possédants une large variété de contraintes temporelles. C'est également un modèle permettant de modéliser un système. 73

SystemC Ensemble de classes $C++$ et de macros qui fournissent une interface de simulation événementielle (*event-driven*). Ces fonctionnalités permettent à un concepteur de simuler des processus concurrents, chacun décrit en utilisant la syntaxe $C++$. Les processus SystemC peuvent communiquer dans un environnement simulé en temps réel, à l'aide de signaux de tous les types de données offerts par le langage $C++$, d'autres offerts par la bibliothèque

SystemC et définis par l'utilisateur. À certains égards, SystemC imite délibérément les langages de description matérielle tel VHDL et Verilog, mais se décrit mieux comme un langage de modélisation au niveau du système (system-level modeling language). SystemC est appliqué à la modélisation au niveau du système, à l'exploration architecturale, à la modélisation des performances, au développement de logiciels, à la vérification fonctionnelle et à la synthèse de haut niveau. SystemC est souvent associé à la conception électronique niveau système (Electronics System-Level - ESL) et à la modélisation au niveau transactionnel (Transaction-Level Modeling - TLM). 14

temps creux Un temps creux peut-être défini comme étant une intervalle de temps Δ au sein de laquelle le traitement processeur est perdu. En effet, si nous considérons les bornes t^- l'instant où le processeur passe en période d'inactivité et t l'instant de réveil du travail le plus proche de t^-, alors durant cette intervalle $[t^-; t]$ aucun travail n'est à traiter. Hors de ces temps creux, le système est en période d'activité. 3

time-driven Une simulation time-driven implique et exécute des évènements à chaque intervalle de temps fixe de Δ unités de temps. Elle se concentre en particulier à vérifier si des évènements ont pu se produire durant cette intervalle de temps fixe. Après l'exécution, le compteur de simulation est avancé de Δ unités de temps et le processus est répété jusqu'à ce que la simulation atteigne un temps de terminaison prédéfini. 3

TLM Le terme Transaction Level Modeling (TLM) désigne une approche de modélisation basée sur les transaction et fondée sur des langages de programmation de haut niveau comme SystemC. Une transaction peut-être définie comme étant un message échangé entre composants du système. Ainsi, la communication est effectuée via l'échange de données sur un protocole abstrait. De manière généraliste, les composants du système sont modélisés comme des modules possédants un ensemble de processus simultanés. Concernant les transactions, celles-ci sont basés sur des méthodes de communication par canaux et la communication est séparée des calculs . 14

WCET Le Worst Case Execution Time (WCET), pire cas de temps d'exécution, équivaut par exemple pour une tâche au temps maximum qu'elle peut prendre à s'exécuter sur une plateforme matérielle spécifique. 16

XML Extensible Markup Language (XML) est un langage balisé conçu pour stocker des informations, communiquer des informations sur un réseau etc. de manière compréhensible par l'humain et la machine facilement. 8

Bibliographie

[1] P. Feiler, B. Lewis and S. Vestal, *The SAE AADL standard : A basis for model-based architecture driven embedded systems engineering*, Workshop on Model-Driven Embedded Systems, 2003.

[2] F. Singhoff, J. Legrand, L. Nana, and L. Marcé, *Cheddar : A Flexible Real Time Scheduling Framework*, International ACM SIGADA Conference, Atlanta, November 2004.

[3] Thomas J. Schriber and Daniel T. Brunner, *How Discrete-Event Simulation Software Works*, *Handbook of Simulation*, Edited by Jerry Banks, John Wiley & Sons, N.Y. 1998, p. 765-811.

[4] Philip Sebastian Kurtin, J.P.H.M. Hausmans, Marco Jan Gerrit Bekooij, *HAPI : An event-driven simulator for real-time multiprocessor systems*, 2016 ACM International Workshop on Software and Compilers for Embedded Systems (SCOPES), New York, 2016, p. 60-66.

[5] Michael González Harbour, J. Javier Gutiérrez José M. Drake, Patricia López Martínez, .J. Carlos Palencia, *Modeling distributed real-time systems with MAST 2*, Journal of Systems Architecture Volume 59, Issue 6, June 2013 (2012), p. 331-340.

[6] Maxime Chéramy, Pierre-Emmanuel Hladik, Anne-Marie Déplanche, *SimSo : A Simulation Tool to Evaluate Real-Time Multiprocessor Scheduling Algorithms*, 5th International Workshop on Analysis Tools and Methodologies for Embedded and Real-time Systems (WATERS), Jul 2014, Madrid, Spain, 6 p., 2014.

[7] Maxime Chéramy, Pierre-Emmanuel Hladik, Anne-Marie Déplanche, Sébastien Dubé, *Simulation of Real-Time Scheduling with Various Execution Time Models*, 9th IEEE International Symposium on Industrial Embedded Systems (SIES), Jun 2014, Pise, Italy.

[8] Y. Chandarli, F. Fauberteau, D. Masson, S. Midonnet, M. Qamhieh, et al. *Yartiss : A tool to visualize, test, compare and evaluate real-time scheduling*

algorithms, 3rd International Workshop on Analysis Tools and Methodologies for Embedded and Real-time Systems, 2012, p. 21–26.

[9] A. M. Uhrmacher. *Dynamic structures in modeling and simulation : a reflective approach, ACM Transactions on Modeling and Computer Simulation*, Vol. 11, No. 2 , April 2001, p. 206-232.

[10] Frank Singhoff, Alain Plantec, Stéphane Rubini, Hai-Nam Tran, Vincent Gaudel, et al.. *Teaching Real-Time Scheduling Analysis with Cheddar*, 9ème édition de l'Ecole d'Eté « Temps Réel », Aug 2015, Rennes, France.

[11] Francis Cottet et Emmanuel Grolleau, *Systèmes temps réel embarqués - 2e éd : Spécification, conception, implémentation et validation temporelle*, Dunod, p.435.

Annexes

A Boucle de simulation non modifiée

```
procedure Build_Multiprocessor_Scheduling
   (Sys                          : in system;
    Result                       : in out Scheduling_Table_Ptr;
    Event_To_Generate            : in Time_Unit_Event_Type_Boolean_Table;
    Last_Time                    : in Natural;
    With_Offsets                 : in Boolean := True;
    With_Precedencies            : in Boolean := True;
    With_Resources               : in Boolean := True;
    With_jitters                 : in Boolean := True;
    With_minimize_preemption     : in Boolean := True;
    With_Task_Specific_Seed      : in Boolean := True;
    Global_Seed_Value            : in Integer := 0;
    Predictable_Global_Seed      : in Boolean := True;
    With_CRPD                    : in Boolean := False)
 is
 begin
    Put_Debug ("Call Build_Multiprocessor_Scheduling", very_verbose);

    initialize_Multiprocessor_Scheduling
      (sys,
       Result,
       Event_To_Generate,
       Last_Time,
       With_Offsets,
       With_Precedencies,
       With_Resources,
       with_jitters,
       With_Task_Specific_Seed,
       Global_Seed_Value,
       Predictable_Global_Seed,
       With_CRPD);

    ------------------------------------------------------------------
    -- Now, we compute the scheduling of the overall architecture
    ------------------------------------------------------------------

    while (Last_Current_Time < Last_Time) loop

       Last_Current_Time := Natural'Last;

       -- Initialized TCB variables that have to be initiliazed
       -- for each unit of time
       --
       for i in 0 .. Si.Number_Of_Tasks - 1 loop
          Si.Tcbs (i).already_run_at_current_time := False;
          si.tcbs (i).wait_for_a_resource:=null;
       end loop;


       for J in 0 .. Result.nb_entries - 1 loop
```

```
            if (Result.entries (J).data.error_msg = empty_string) then

               if (Current_Time (J) < Last_Time) then

                  Put_Debug("Build_Multiprocessor_Scheduling : compute
↪  scheduling of time " &
                                 To_Unbounded_String (Natural'Image
↪  (Current_Time (J))),very_verbose);

                  for core_id in 0 .. table_of_core_scheduler(j).nb_entries -
↪  1 loop

                     -- Preemptive case : if a task has been previously
                     -- elected and if
                     -- it has not ended its work => re-elect it !
                     --
                     must_perform_election := True;
                     if (Get_Preemptive(table_of_core_scheduler(j).entries
↪  (core_id).scheduler.all) =
                           not_preemptive) then

                        if (table_of_core_scheduler(j).entries
↪  (core_id).scheduler.previous_running_task_is_not_completed) then
                           Put_Debug ("Preemption management : task is not
↪  ended ", very_verbose);
                           Elected               :=
↪  table_of_core_scheduler(j).entries
↪  (core_id).scheduler.Previously_Elected;
                           No_Task               := False;
                           must_perform_election := False;
                        end if;
                     end if;

                     if must_perform_election then
                        Put_Debug ("Call Do_Election ", very_verbose);
                        Do_Election
                          (table_of_core_scheduler(j).entries
↪  (core_id).scheduler.all,
                           Si,
                           Result.entries (J).data.result,
                           Result.entries (J).data.scheduling_msg,
                           Current_Time (J),
                           Result.entries (J).item.name,
                           To_Unbounded_String (""),
                           sys.Dependencies,
                           With_Offsets,
                           With_Precedencies,
                           With_Resources,
                           With_jitters,
                           With_minimize_preemption,
                           Event_To_Generate,
                           Elected,
                           No_Task);
                     end if;
```

```
                     -- We have a task to run
                     --
                     if not No_Task then
                        Put_Debug("Call
↪  Update_Task_Simulation_Properties_And_Produce_Events ",very_verbose);

                        -- Update task properties and produce events
                        --
                        Update_Task_Simulation_Properties_And_Produce_Events
                          (table_of_core_scheduler(j).entries
↪  (core_id).scheduler.all,
                           Result.entries (J).item.name,
                           Si,
                           sys.Dependencies,
                           Elected,
                           Result.entries (J).data.result,
                           Current_Time (J),
                           Last_Time,
                           With_Offsets,
                           With_Precedencies,
                           With_Resources,
                           with_jitters,
                           Event_To_Generate,
                           With_CRPD);
                     end if;
                  end loop;

                  Current_Time (J) := Current_Time (J) + 1;

                  Last_Current_Time := Natural'Min (Last_Current_Time,
↪  Current_Time (J));
               end if;
            end if;
         end loop;
      end loop;

   end Build_Multiprocessor_Scheduling;
```

B Commandes de contrôle de Cheddar

```
   --------------------------------
   -- Start_State_Communication --
   --------------------------------

   procedure AADLInspector_Command_Start_State_Communication (
     Slice_Size       : in out Natural;
     Last_Time_Mod    : in out Natural;
     Speed            : in out Duration;
     SpeedFactor      : in     Duration;
     Exit_Simulation  : in out Boolean)
   is
     Message_Received : Unbounded_String;
     Message_To_Send  : Unbounded_String;
   begin
     -- Initialization ended, waiting for a socket message:
     Message_Received := Sockets_Overlay.Read(Command_Socket,
↪  Command_Channel);

     Put_Debug("__INFO__ :: Message_Received:" & Message_Received);
     while(Index(Message_Received, "simulator play") <= 0) loop
       if (Index(Message_Received, "slice") > 0) then
         Slice_Size    := Integer'Value
↪  (Get_Command_Value(Message_Received));
         Last_Time_Mod := Slice_Size;
       elsif (Index(Message_Received, "speed") > 0) then
         Message_To_Send := To_Unbounded_String("Change time between tick OK
↪  :*");
         Write_Channel (Ack_Channel, Message_To_Send & Character'Val(10));
         Speed := SpeedFactor * Duration'Value
↪  (Get_Command_Value(Message_Received));
       elsif (Index(Message_Received, "get_tasks_capacities") > 0) then
         Put_Debug("__INFO__ :: Request for table of tasks capacities");
         -- Message_To_Send := Encode_Task_Capacities(Si);
       end if;
       Message_Received := Sockets_Overlay.Read(Command_Socket,
↪  Command_Channel);
     end loop;
   end AADLInspector_Command_Start_State_Communication;
```

```
   ------------------------------
   -- Run_State_Communication --
   ------------------------------

   procedure AADLInspector_Command_Run_State_Communication (
     Slice_Size       : in out Natural;
     Last_Time_Mod    : in out Natural;
     Speed            : in out Duration;
     SpeedFactor      : in     Duration;
     Exit_Simulation  : in out Boolean)
   is
     Message_Received : Unbounded_String;
     Message_To_Send  : Unbounded_String;
   begin
     Message_Received := Sockets_Overlay.Read(Command_Socket,
↪  Command_Channel);
     if(To_String(Message_Received) = "simulator pause") then
       Message_To_Send := To_Unbounded_String("pause OK");
       Write_Channel (Ack_Channel, Message_To_Send & Character'Val(10));
       while(To_String(Message_Received) /= "simulator resume") loop
         Message_Received := Sockets_Overlay.Read(Command_Socket,
↪  Command_Channel);
         if (Index(Message_Received, "slice") > 0) then
           Slice_Size := Integer'Value
↪  (Get_Command_Value(Message_Received));
           Last_Time_Mod := Last_Time_Mod + Slice_Size;
         elsif (Index(Message_Received, "speed") > 0) then
           Message_To_Send := To_Unbounded_String("Change time between tick
↪  OK :*");
           Write_Channel (Ack_Channel, Message_To_Send & Character'Val(10));
           Speed := SpeedFactor * Duration'Value
↪  (Get_Command_Value(Message_Received));
         end if;
       end loop;
       Message_To_Send := To_Unbounded_String("resume OK");
       Write_Channel (Ack_Channel, Message_To_Send & Character'Val(10));
     elsif (Index(Message_Received, "slice") > 0) then
       Slice_Size := Integer'Value (Get_Command_Value(Message_Received));
       Last_Time_Mod := Last_Time_Mod + Slice_Size;
     elsif (Index(Message_Received, "speed") > 0) then
       Message_To_Send := To_Unbounded_String("Change time between tick OK
↪  :*");
       Write_Channel (Ack_Channel, Message_To_Send & Character'Val(10));
       Speed := SpeedFactor * Duration'Value
↪  (Get_Command_Value(Message_Received));
     elsif To_String(Message_Received) = "simulator stop" then
↪  Exit_Simulation := True; end if;
   end AADLInspector_Command_Run_State_Communication;
```

```
   ------------------------------
   -- End_State_Communication --
   ------------------------------

   procedure AADLInspector_Command_End_State_Communication (
     Slice_Size       : in out Natural;
     Last_Time_Mod    : in out Natural;
     Speed            : in out Duration;
     SpeedFactor      : in     Duration;
     Exit_Simulation  : in out Boolean)
   is
     Message_Received : Unbounded_String;
     Message_To_Send  : Unbounded_String;
   begin
     Message_Received := Sockets_Overlay.Read(Command_Socket,
↪  Command_Channel);
     while(To_String(Message_Received) /= "simulator play") loop
       if (Index(Message_Received, "slice") > 0) then
         Slice_Size := Integer'Value (Get_Command_Value(Message_Received));
       end if;
       if (Index(Message_Received, "speed") > 0) then
         Message_To_Send := To_Unbounded_String("Change time between tick OK
↪  :*");
         Write_Channel (Ack_Channel, Message_To_Send & Character'Val(10));
         Speed := SpeedFactor * Duration'Value
↪  (Get_Command_Value(Message_Received));
       end if;
       if To_String(Message_Received) = "simulator stop" then
↪  Exit_Simulation := True; end if;
         exit when To_String(Message_Received) = "simulator stop";
       Message_Received := Sockets_Overlay.Read(Command_Socket,
↪  Command_Channel);
     end loop;

     Last_Time_Mod := Last_Time_Mod + Slice_Size;
   end AADLInspector_Command_End_State_Communication;
```

C Boucle de simulation modifiée

```
   ----------------------------------------------------------------------
   -- Build_Multiprocessor_Scheduling
   -- Purpose: Calculations in order to build the scheduling simulation
   --          (Time-Driven) and send at each unit of time the scheduling
   --          informations to a data socket (AADLInspector). Moreover,
   --          the simulation can be stopped/accelerated/paused/played
   --          by a command socket (AADLInspector command to Cheddar)
   -- Extra: -Currently two packages are used for handling
   --
   --          1) The sockets and stuff linked to their manipulation:
↪  Sockets_Overlays
   --          2) The manipulations/translation of events strings for
   --             AADLInspector / Marzhin replacement: Marzhin_Utils
   ----------------------------------------------------------------------
  procedure Build_Multiprocessor_Scheduling
    (Sys                     : in     system;
    Result                   : in out Scheduling_Table_Ptr;
    Event_To_Generate        : in     Time_Unit_Event_Type_Boolean_Table;
    Last_Time                : in     Natural;
    With_Offsets             : in     Boolean := True;
    With_Precedencies        : in     Boolean := True;
    With_Resources           : in     Boolean := True;
    With_jitters             : in     Boolean := True;
    With_minimize_preemption : in     Boolean := True;
    With_Task_Specific_Seed  : in     Boolean := True;
    Global_Seed_Value        : in     Integer := 0;
    Predictable_Global_Seed  : in     Boolean := True;
    With_CRPD                : in     Boolean := False)
  is
    Speed               : Duration          := 1.0;       -- The time delay
↪   between each time unit calculation (seconds)
    SpeedFactor         : constant Duration := 0.001;     -- The speed factor
↪   used to calculate delay (SpeedValue * SpeedFactor)
    Exit_Simulation     : Boolean           := False;     -- True if
↪   simulation received "stop" command
    Slice_Size          : Natural           := Last_Time; -- Size of the
↪   simulation time slice
    Last_Time_Mod       : Natural           := Last_Time; -- End simulation
↪   time unit
  begin
```

```
   Put_Debug ("__DEBUG__ :: Call Build_Multiprocessor_Scheduling",
↪  very_verbose);

   if Socket_Mode_Activated then
     Initialize_Sockets;
     AADLInspector_Communication_State := Starting;
     AADLInspector_Command_Communication (Slice_Size, Last_Time_Mod, Speed,
↪  SpeedFactor, Exit_Simulation);
   end if;

   Put_Debug("__INFO__ :: Starting simulation up to " & Last_Time_Mod'Img,
↪  very_verbose);

   initialize_Multiprocessor_Scheduling
     (sys,
      Result,
      Event_To_Generate,
      Last_Time_Mod,
      With_Offsets,
      With_Precedencies,
      With_Resources,
      with_jitters,
      With_Task_Specific_Seed,
      Global_Seed_Value,
      Predictable_Global_Seed,
      With_CRPD);

   ------------------------------------------------------------------
   -- Now, we compute the scheduling of the overall architecture
   ------------------------------------------------------------------
   Process_Each_Time_Unit:
   while (Last_Current_Time < Last_Time_Mod + 2) loop
     Last_Current_Time := Natural'Last;

     ------------------------------------------------------------------
     -- Initialize TCB variables
     ------------------------------------------------------------------
     for z in 0 .. Si.Number_Of_Tasks - 1 loop
       Si.Tcbs (z).already_run_at_current_time := False;
       si.tcbs (z).wait_for_a_resource         := null;
     end loop;
```

```
      Process_Each_Result_Entry:
      for J in 0 .. Result.nb_entries - 1 loop
        ------------------------------------------------------------------
        -- Check time and errors conditions
        ------------------------------------------------------------------
        if  (Result.entries (J).data.error_msg /= empty_string)
        and (Current_Time (J) >= Last_Time_Mod + 2)
        then
          goto End_Of_Entries_Loop;
        end if;

        Put_Debug ("__INFO__ :: Last_Time : " & Last_Time'Img);
        Put_Debug ("__INFO__ :: Build_Multiprocessor_Scheduling : compute
↪   scheduling of time " &
                   To_Unbounded_String (Natural'Image (Current_Time (J))),
                   very_verbose);

        Process_Each_Core:
        for core_id in 0 .. table_of_core_scheduler(j).nb_entries - 1 loop
          ------------------------------------------------------------------
          -- Check if re-election of previous running task's needed
          ------------------------------------------------------------------
          must_perform_election := True;
          if  (Get_Preemptive (table_of_core_scheduler(j).entries
↪   (core_id).scheduler.all) = not_preemptive) then
            if (table_of_core_scheduler(j).entries
↪   (core_id).scheduler.previous_running_task_is_not_completed)
            then
              Put_Debug ("__INFO__ :: Preemption management : task is not
↪   ended ", very_verbose);
              Elected := table_of_core_scheduler(j).entries
↪   (core_id).scheduler.Previously_Elected;
              No_Task               := False;
              must_perform_election := False;
            end if;
          end if;
```

```
         if must_perform_election then
--#[debug]
            Put_Debug ("__DEBUG__ :: Call Do_Election ", very_verbose);
            Do_Election
              (table_of_core_scheduler(j).entries (core_id).scheduler.all,
               Si,
               Result.entries (J).data.result,
               Result.entries (J).data.scheduling_msg,
               Current_Time (J),
               Result.entries (J).item.name,
               To_Unbounded_String (""),
               sys.Dependencies,
               With_Offsets,
               With_Precedencies,
               With_Resources,
               With_jitters,
               With_minimize_preemption,
               Event_To_Generate,
               Elected,
               No_Task);
         end if;

         if not No_Task then
--#[debug]
            Put_Debug ("__DEBUG__ :: Call
↪  Update_Task_Simulation_Properties_And_Produce_Events ", very_verbose);

            Update_Task_Simulation_Properties_And_Produce_Events
              (table_of_core_scheduler(j).entries (core_id).scheduler.all,
               Result.entries (J).item.name,
               Si,
               sys.Dependencies,
               Elected,
               Result.entries (J).data.result,
               Current_Time (J),
               Last_Time_Mod,
               With_Offsets,
               With_Precedencies,
               With_Resources,
               with_jitters,
               Event_To_Generate,
              With_CRPD);
         end if;
```

```
         if Optimization_Mode_Activated then
           ------------------------------------------------------------------
           -- Computing of the idle times based heuristic
           ------------------------------------------------------------------
           First_Heuristic:
           declare
             Jump_Time : Natural; -- local variable used only for first
↪  heuristic calculations.
           begin
           Update_Values (Heuristic_1(Idle_Heuristic.all), No_Task,
↪  Natural(core_id));
           if Heuristic_1(Idle_Heuristic.all).Number_Of_Valid_Idle_Times =
↪  Natural(table_of_core_scheduler(j).nb_entries) then
             Jump_Time := Calculate_Heuristic (Idle_Heuristic.all, Si);
             if Jump_Time > Current_Time (J) + 2 then
               Heuristic_1(Idle_Heuristic.all).Current_Time :=
↪  Current_Time(J);
               if table_of_core_scheduler(j).entries (core_id).scheduler.all
↪  in Hierarchical_* then
                 Increase_Used_CPU(Hierarchical_*(table_of_core_scheduler(j)
                 .entries (core_id).scheduler.all),
↪  (Jump_Time-Current_Time(J)-1));
                 Jump_Time := Jump_Time - 1;
               end if;
               Current_Time (J)  := Jump_Time;
               Last_Current_Time := Current_Time (J);
               Reset_Values (Heuristic_1(Idle_Heuristic.all),
↪  Natural(core_id));
               goto BACK_TO_THE_FUTURE;
             end if;
           end if;
           end First_Heuristic;

           ------------------------------------------------------------------
           -- Computing of the last job running based heuristic
           ------------------------------------------------------------------
           Heuristic_2(Last_Job_Heuristic.all).Current_Time :=
↪  Current_Time(J);
           if Verify_Pre_Conditions(Heuristic_2(Last_Job_Heuristic.all), Si)
↪  then
             for RC in 0 .. Si.Tcbs(Heuristic_2(Last_Job_Heuristic.all)
             .Remaining_Task_Id).Rest_Of_Capacity - 1 loop
               Generate_Running_Event(table_of_core_scheduler(j).entries
↪  (core_id).
               scheduler.all, Si, Result.entries (J).data.result,
               Elected, Current_Time (J) + RC);
             end loop;
             Current_Time(J)   :=
↪  Calculate_Heuristic(Last_Job_Heuristic.all, Si);
             Last_Current_Time := Current_Time (J);
             Reset_Values(Heuristic_2(Last_Job_Heuristic.all), Si);
             goto BACK_TO_THE_FUTURE;
           end if;
         end if;
```

```
         ------------------------------------------------------------
         -- Send events data to AADLInspector
         ------------------------------------------------------------
         if Socket_Mode_Activated then
           AADLInspector_Data_Communication(Sys, Result, J, Current_Time(J),
↪  Last_Time_Mod, SpeedFactor, Speed, Slice_Size, Exit_Simulation);
         end if;

         if Exit_Simulation then goto End_Of_Simulation; end if;
       end loop Process_Each_Core;

       Current_Time (J)  := Current_Time (J) + 1;
       Last_Current_Time := Natural'Min (Last_Current_Time, Current_Time
↪  (J));
       <<Back_To_The_Future>>

       -- Check if simulation reached it's last time unit:
       if (Current_Time(J) = Last_Time_Mod + 2) and Socket_Mode_Activated
↪  then
         AADLInspector_Communication_State := Ending;
         AADLInspector_Command_Communication (Slice_Size, Last_Time_Mod,
↪  Speed, SpeedFactor, Exit_Simulation);
       end if;

       if Exit_Simulation then goto End_Of_Simulation; end if;

       <<End_Of_Entries_Loop>>
     end loop Process_Each_Result_Entry;
   end loop Process_Each_Time_Unit;

   <<End_Of_Simulation>>
   Put_Debug("__INFO__ :: End of simulation");
   if Socket_Mode_Activated then Close_Sockets; end if;
 end Build_Multiprocessor_Scheduling;
```

D Code méthodes de l'heuristique 2

```
   ---------------------------------------
   -- Heuristic_2::Get_Remaining_Task --
   ---------------------------------------

   function Get_Remaining_Task (
     This         : in out Heuristic_2;
     Si           : in     Scheduling_Information;
     Current_Time : in     Natural)
   return Boolean is
   begin
     This.Number_Of_Tasks_Ended := 0;
     for num_task in 0 .. Si.Number_Of_Tasks - 1 loop
       if (Si.Tcbs(num_task).Wake_Up_Time > Current_Time) and
          (Si.Tcbs(num_task).Rest_Of_Capacity =
↪    Si.Tcbs(num_task).Tsk.capacity)
       then
         This.Number_Of_Tasks_Ended := This.Number_Of_Tasks_Ended + 1;
       else
         This.Remaining_Task_Id := num_task;
       end if;
     end loop;

     if This.Number_Of_Tasks_Ended = Natural(Si.Number_Of_Tasks) - 1 then
       return True;
     end if;

     return False;
   end Get_Remaining_Task;
```

```
   ------------------------------------------
   -- Heuristic_2::Get_Nearest_Wake_Up_Time --
   ------------------------------------------

   function Get_Nearest_Wake_Up_Time (
     This              : in out Heuristic_2;
     Si                : in Scheduling_Information)
   return Natural is
     Nearest_Wake_Up_Time : Natural := This.Current_Time +
↪    Si.Tcbs(This.Remaining_Task_Id).Rest_Of_Capacity;
   begin
     for num_task in 0 .. Si.Number_Of_Tasks - 1 loop
       if Si.Tcbs(num_task).Tsk.name /=
↪    Si.Tcbs(This.Remaining_Task_Id).Tsk.name then
          if Si.Tcbs(num_task).Wake_Up_Time < Nearest_Wake_Up_Time then
              Nearest_Wake_Up_Time := Si.Tcbs(num_task).Wake_Up_Time;
          end if;
       end if;
     end loop;

     return Nearest_Wake_Up_Time;
   end Get_Nearest_Wake_Up_Time;
```

```
 -------------------------------
 -- Heuristic_2::Reset_Values --
 -------------------------------

 procedure Reset_Values (This : in out Heuristic_2; Si : in
↪  Scheduling_Information) is
 begin
   if (Si.Tcbs(This.Remaining_Task_Id).Rest_Of_Capacity > 1) then
     This.Commit_Unit := Si.Tcbs(This.Remaining_Task_Id).Rest_Of_Capacity -
↪  2;
   else
     This.Commit_Unit := 0;
     Put_Debug("INFO SAFETY PUT TO ZERO");
   end if;

   Heuristic_2(Last_Job_Heuristic.all).Remaining_Task_Id     := 0;
   Heuristic_2(Last_Job_Heuristic.all).Number_Of_Tasks_Ended := 0;
 end Reset_Values;
```

```
 ----------------------------
 -- Heuristic_2::Calculate --
 ----------------------------

 function Calculate (This : in out Heuristic_2; Si : in
↪  Scheduling_Information) return Natural is
   Jump_Time : Natural;
   Nearest_Wake_Up_Time : Natural;
 begin
   Jump_Time := This.Current_Time +
↪  Si.Tcbs(This.Remaining_Task_Id).Rest_Of_Capacity;
   Nearest_Wake_Up_Time := Get_Nearest_Wake_Up_Time(This, Si);

   Si.Tcbs(This.Remaining_Task_Id).Rest_Of_Capacity := Jump_Time -
↪  Nearest_Wake_Up_Time + 1;
   Si.Tcbs(This.Remaining_Task_Id).Used_Capacity :=
↪  Si.Tcbs(This.Remaining_Task_Id).Tsk.capacity - (Jump_Time -
↪  Nearest_Wake_Up_Time) - 1;

   if Nearest_Wake_Up_Time < Jump_Time then
     Jump_Time := Nearest_Wake_Up_Time;
   end if;

   return Jump_Time;
 end Calculate;
```

E Résultats des tests d'implémentation pour Cheddar

Test	H1	H2
EDF_NP_FM		
EDF_NP_JLM		
EDF_P_FM		
EDF_P_JLM		
EDF_P_JLM_1		
gEDF_vs_EDZL1		
gEDF_vs_EDZL2		
global_scheduling1		
global_scheduling2		
global_scheduling3		
global_scheduling4		
LLF_NP_FM		
LLF_NP_JLM		
LLF_P_FM		
LLF_P_FM_1		
LLF_P_JLM		
RM_NP_FM		
RM_NP_FM_1		
RM_NP_JLM		
RM_P_FM		
RM_P_FM_1		
RM_P_JLM		
aperiodic0		
Hiearchical_cycle1		
Hierarchical_offline1		
Hierarchical_offline2		
Hierarchical_offline3		
jitter_1		
jitter_2		

H1	Saut sur temps creux
H2	Saut sur job si seul ce job en exécution

	Screenshot **identique** à l'exécution du Cheddar modifié.
	Screenshot **non identique** à l'exécution du Cheddar modifié et **non identique** à l'exécution du Cheddar non modifié. **Mais exécution identique** entre Cheddar non modifié et Cheddar modifié.
	Fichier xmlv3 ne correspond pas aux données du screenshot et xml associé non trouvé.

F Modifications apportées à Marzhin

```
-----------------------------------------
>SimpleThread
/**
   * Traitement avant tick.
   * ---------------------------------------
   * Heuristique 1
   * ---------------------------------------
   * TODO:
   *   - Remettre HashMap de monitoring en <String, Boolean>
   *   - Tests comparaison Cheddar
   * */
  public void beginTick()
  {
    //
    // Traitement eventuel sur les sous-composants en debut de cycle.
    for (IEntity loEntity : coChildren) {
      loEntity.beginTick();
    }

    //
    // Reinitialisation des erreurs
    coErrors.clear();

    //
    // Gestion des Transitions/Etapes.
    switch (cEDispatchTransition) {
      case TRANSITION_Complete: {

        //
        // gestion etat.
        setState(THREAD_STATE_SUSPENDED);
        setPreviousState(getState());

        ciRealDuration = 0;
        ciQuantumDuration = 0;
        if (!(coDispatchProtocol instanceof Background)) {
          if (ciQuantum!=Integer.MIN_VALUE) {
            if (ciQuantumEndCounter==0) {
              computeQuantumPriority();
            }

            final String lSKey =
↪   String.valueOf((double)((int)getPriority()));
            if
↪   (!"RANDOM".equals(getProcess().getQuantumDispatchOrder().get(lSKey)))  {
              if (getProcess().getQuantumDispatchOrder().containsKey(lSKey) )
↪   {
                ArrayList<SimpleThread> loThreads =
↪   getProcess().getThreadsWithQuantum().get(lSKey);
```

```
                boolean lbQuantumAllComplete = true;
                for (SimpleThread loThread : loThreads) {
                  if (loThread.getState()!=THREAD_STATE_SUSPENDED) {
                    lbQuantumAllComplete = false;
                  }
                }
                if (lbQuantumAllComplete) {
                  loThreads.clear();
                  for (SimpleThread loThread :
 ↪  getProcess().getInitOrderThreadsWithQuantum().get(lSKey)) {
                    loThreads.add(loThread);
                  }
                  final int size = loThreads.size();
                  for (int i=0;i<size;++i) {
//                    System.out.print("SimpleThread::beginTick SETPRIORITY "
 ↪  + loThreads.get(i).getName() + " " + loThreads.get(i).getPriority());
                    loThreads.get(i).setPriority(Double.parseDouble(lSKey) +
 ↪  (double)i/10.0);
//                    System.out.println(" -> " +
 ↪  loThreads.get(i).getPriority());
                  }
                }
              }
            }
            ciQuantumEndCounter = 0;
          }
        }

        /** Première heuristique (temps creux) **/
        if(this.getType().equals("THREAD") && true){
          SystemRoot loSystemRoot = (SystemRoot)Entity.getRoot();
          loSystemRoot.setMonitoredThreadActivityState(this.getId(), false);

          if(loSystemRoot.jumpConditionsValid())

 ↪  Scheduler.getScheduler().setCurrentTick(loSystemRoot.getJumpTimeUnit());
        }

        coStoredPortInValues.clear();
        cbJustComplete = false;
        break;
      }
```

```
-------------------------------------------
>SimpleThread:
  public void setJustDispatch(boolean pbJustDispatch) {
    cbJustDispatch = pbJustDispatch;
    if (pbJustDispatch) {
      fireEvent("THREAD_DISPATCH" +
 ↪   getDispatchProtocol().getDispatchMessage());
      if(this.getType().equals("THREAD"))

 ↪   ((SystemRoot)Entity.getRoot()).setMonitoredThreadActivityState(this.getId(),
 ↪   true);
    }
  }

-------------------------------------------
package com.virtualys.ellidiss.entity.thread;

/**
 * Cette classe permet de stocker
 * les données d'un thread ainsi que
 * des données supplémentaires lui
 * étant liés. Utilisé pour les
 * heuristiques de simulation
 * hybrid-driven.
 *
 *
 * Actuellement,
 * Seule Heuristique 1
 * (Temps creux) implémentée.
 *
 */
public class ThreadData {
  public SimpleThread thread;

  /**
   * Le thread est actif (true)
   * Le thread est inactif [complet](false)
   */
    public Boolean active;

    public ThreadData(SimpleThread thread){
      this.thread = thread;
      this.active = true;
    }

    public ThreadData(SimpleThread thread, Boolean active){
      this.thread = thread;
      this.active = active;
    }
}
```

```
-----------------------------------------
>SystemRoot:
    /* Activation des optimisations. false[désactif] */
  protected boolean opt1 = false;

  public boolean getOpt1(){
    return opt1;
  }

  public void setOpt1(boolean val){
    opt1 = val;
  }



  /**
   * Contient des données supplémentaires utiles
   * aux heuristiques concernant les threads du
   * système. (ex: état de complétude des threads)
   *
   * Clé    : Id du thread
   * Valeur : ThreadData: données du thread +
   *                      données supplémentaires.
   *
   * @see  ThreadData
   */
    public HashMap<String, ThreadData> threadMonitoringMap = new
↪    HashMap<String, ThreadData>();

    /**
     * Ajoute un thread au HashMap permettant de
     * monitorer les threads et leurs données utiles aux
     * heuristiques.
     *
     * @param thread
     */
  public void addMonitoredThread(SimpleThread thread){
    threadMonitoringMap.put(thread.getId(), new ThreadData(thread, true));
  }
```

```
/**
 * Met à jour l'état d'activité du thread passé
 * en paramètre.
 *
 * @param threadId  Id du thread
 * @param isActive  État d'activité du thread.
 */
public void setMonitoredThreadActivityState(String threadId, Boolean
↪  isActive){
  ThreadData newThreadData = threadMonitoringMap.get(threadId);
  newThreadData.active = isActive;
  threadMonitoringMap.replace(threadId, newThreadData);
}

/**
 * Vérifie que l'ensemble des threads ont
 * consommés l'ensemble de leurs capacités
 *
 * @return
 *     true si threads terminés, false sinon.
 */
public boolean isAllMonitoredThreadsCompleted(){
  for (ThreadData thread : threadMonitoringMap.values())
    if (thread.active) return false;
  return true;
}

/**
 * Vérifie si il reste un seul thread
 * n'ayant pas consommé l'ensemble
 * de sa capacité.
 *
 * @return
 *     true si un seul thread restant, false sinon.
 */
public boolean isOnlyOneThreadRemaining(){
  int activeThreads = 0;

  for (ThreadData thread : threadMonitoringMap.values()){
    if (thread.active) activeThreads++;
    if (activeThreads > 1) return false;
  }
  return (activeThreads == 1) ? true : false;
}
```

```
/**
 * Retourne l'Id du dernier thread non
 * terminé (capacité non consommée).
 *
 * @return
 *     Id du thread.
 */
public String getRemainingThreadId(){
  if (!isOnlyOneThreadRemaining()) return null;
  for (Map.Entry entry : threadMonitoringMap.entrySet())
    if (((ThreadData)entry.getValue()).active == true)
      return (String) entry.getKey();
  return null;
}

/**
 * Retourne l'unité de temps jusqu'à
 * laquelle l'ordonnanceur peut
 * "sauter".
 *
 * @return
 *     L'unité de temps future.
 */
public long getJumpTimeUnit(){
  long llNearestDispatchTime = Long.MAX_VALUE;

  for (Map.Entry<String, Entity> entry :
↪  SimpleThread.getMapEntities().entrySet())
      if(entry.getValue().getType().equals("THREAD")){
        SimpleThread st       = (SimpleThread) entry.getValue();
        DispatchProtocol dp   = (DispatchProtocol)st.getDispatchProtocol();
        llNearestDispatchTime = (dp.getNextDispatchTick() <
↪  llNearestDispatchTime &&
                                dp.getNextDispatchTick() >
↪  Scheduler.getScheduler().getCurrentTick())
                    ? dp.getNextDispatchTick() : llNearestDispatchTime;
      }
  return llNearestDispatchTime;
}
```

```
/**
 * Vérifie les différentes pré-conditions de
 * l'heuristique.
 *
 * @return
 *     true si conditions valides, false sinon.
 */
public boolean jumpConditionsValid(){
  if (!this.isAllMonitoredThreadsCompleted())
    return false;

  boolean wrongDispatchTime = false;

  for(ThreadData threadData : this.threadMonitoringMap.values()){
    int eventQueueSize  = 0;
    SimpleThread thread = threadData.thread;
    String dispatchProtocolName =
↪   thread.getDispatchProtocol().getClass().getSimpleName();

    if(dispatchProtocolName.equals("Timed")){
      eventQueueSize =
↪   ((Timed)thread.getDispatchProtocol()).coEventsFromPort.size();
    }else if(dispatchProtocolName.equals("Hybrid")){
      if( ((Periodic)thread.getDispatchProtocol()).getNextDispatchTick() ==
↪   Scheduler.getScheduler().getCurrentTick() )
        wrongDispatchTime = true;
      eventQueueSize =
↪   ((Hybrid)thread.getDispatchProtocol()).coEventsFromPort.size();
    }else if(dispatchProtocolName.equals("Sporadic")){
      eventQueueSize =
↪   ((Sporadic)thread.getDispatchProtocol()).coEventsFromPort.size();
    }else if(dispatchProtocolName.equals("Periodic")){
      if( ((Periodic)thread.getDispatchProtocol()).getNextDispatchTick() ==
↪   Scheduler.getScheduler().getCurrentTick() )
        wrongDispatchTime = true;
    }else if(dispatchProtocolName.equals("Aperiodic")){
      eventQueueSize =
↪   ((Aperiodic)thread.getDispatchProtocol()).coEventsFromPort.size();
    }

    if(eventQueueSize > 0 || wrongDispatchTime){ return false; }
  }
  return true;
}
```

```
-------------------------------------------
>Communication:
public synchronized void processEventReceived(CommunicationEvent poEvent) {
  try {

    //[...]

    if("set_option".equals(lSCommandName)){
                try {
                  if (lSCommandValue.equals("opt1")){
                    ((SystemRoot)Entity.getRoot()).setOpt1(true);
                    writeAcknowledge("Change option OK : " + lSCommandValue);
                    return;
                  }
                } catch (java.lang.NumberFormatException ex) {
                 }
        return;
             }

        //[...]
    }
}
-------------------------------------------
>CommunicationEvent:
public CommunicationEvent(ICommunicationChannel poCommunication, String
 ↪  pSMessage){
  super(poCommunication);
  cSMessage = pSMessage;
  try {
    String[] st = cSMessage.split(" ");
    //
    // tick programme eventuel.
    if (st.length>2 && st[1].contains("tick=")) {
      cLTick = Long.parseLong(st[1].substring("tick=".length()));
      cSMessage = st[0];
      for (int i=2;i<st.length;++i) {
        cSMessage += " " + st[i];
      }
      st = cSMessage.split(" ");
    }
    //
    //
```

```
    if (st.length>0) {
      if ("device".equals(st[0]) ||
        "send_error".equals(st[0]) ||
        "send_state".equals(st[0]) ||
        "set_timetick".equals(st[0]) ||
        "get_timetick".equals(st[0]) ||
        "get_entity_all_properties".equals(st[0]) ||
        "get_entity_properties".equals(st[0]) ||
        "set_entity_property".equals(st[0]) ||
        "set_text_display_padding".equals(st[0]) ||
        "set_text_display_exec_unit".equals(st[0]) ||
        "get_ports".equals(st[0]) ||
        "get_devices".equals(st[0]) ||
        "set_current_tick".equals(st[0])||
        "set_option".equals(st[0])) { //ICI

        ceType = Type.COMMON_CMD;
      } else {
        //[...]
      }
  //[...]
}
-----------------------------------------
>System:
@Override
public void startElement(String pSUri, String pSLocalName, String pSName,
 ↪  Attributes poAttributes)
  throws SAXException, NumberFormatException{
  if ("property".equals(pSName)) {
    final String lSName = poAttributes.getValue("name");
    final String lSValue = poAttributes.getValue("value");
    if (!"".equals(lSName) && !"".equals(lSValue)) {
      if ("option".equals(lSName.toLowerCase())) {
              if(lSValue.equals("opt1"))
                ((SystemRoot)Entity.getRoot()).setOpt1(true);
            }
          addProperty(lSName, lSValue);
    }
  }
//[...]
}
-----------------------------------------
```

Résumé

Ce rapport résume le travail effectué durant mon stage de $2^{ème}$ année de Master en Logiciels pour Systèmes Embarqués de l'Université de Brest Occidentale. Celui-ci d'une durée de 5 mois portait sur l'étude d'optimisation de simulation d'ordonnancement, problématique à laquelle une solution est proposée, basée sur une approche hybride entre les approches time-driven et event-driven. Cette solution sera intégrée au logiciel de simulation Cheddar qui servira de cas d'étude. Le logiciel de simulation Marzhin servira quant à lui de cas pratique.