ÉCOLE UNIVERSITAIRE
DE PHYSIQUE ET D'INGÉNIERIE
Université Clermont Auvergne

Capgemini

MEMOIRE DE STAGE

2ème année de Master mention Automatique Robotique

Option : Mécatronique

Conception d'une plateforme de co-simulation basée sur le standard FMI

Réalisé par :
M. EL AMRANI Mouad

Encadré par :
Mme. SHOURICK Hélèna (Cap)
M. CHATEAU Thierry (UCA)

Soutenu le 03 Septembre 2024, Devant le jury composé de :

M. TRASSOUDAINE Laurent :	EUPI	- Jury
M. CHATEAU Thierry :	UCA	- Encadrant académique
Mme. SHOURICK Hélèna :	Capgemini	- Encadrant professionnel

Promotion : 2023/2024

Résumé

La recherche de nouvelles approches de collaboration interdisciplinaire est essentielle pour naviguer dans les complexités du développement de systèmes face à des exigences de marché en augmentation. Une approche envisageable pour surmonter ce défi est l'adoption d'une méthode basée sur des modèles hétérogènes. Cette méthode permet à diverses équipes de créer leurs propres modèles et de conduire leurs analyses habituelles dans leur discipline respective. De plus, elle offre la possibilité de coupler ces modèles pour réaliser des simulations conjointes (co-simulation), facilitant ainsi l'analyse du comportement global du système. La norme Functional Mock-up Interface (FMI), développée par l'association Modelica, joue un rôle crucial dans la facilitation de cette intégration. FMI fournit un cadre polyvalent pour l'échange aisé de modèles et de données de simulation entre différents outils et plates-formes.

Dans ce travail, nous présentons une approche globale de la taxonomie de la co-simulation, en utilisant des notions ontologiques pour segmenter en définissant systématiquement les divers éléments et interactions au sein des environnements de co-simulation. Cette taxonomie structurée sert de base pour améliorer la clarté et l'interopérabilité des processus de co-simulation. En outre, nous nous concentrons sur l'optimisation de l'orchestrateur d'OMSimulator, une plateforme conçue pour la co-simulation basée sur le standard FMI. Pour ce faire, nous avons employé une méthode itérative appelée méthode d'Aitken-Schwarz. En intégrant la méthode d'Aitken-Schwarz dans l'orchestrateur d'OMSimulator, nous avons pu améliorer significativement la stabilité des simulations numériques, réduire les erreurs et assurer une convergence plus rapide des résultats.

Ces améliorations permettent de renforcer les processus de co-simulation, offrant ainsi une solution plus robuste et efficace pour le développement de systèmes complexes dans un environnement interdisciplinaire.

Mots clés : Co-simulation, FMI, Taxonomie, OMSimulator, Aitken-Schwarz.

Abstract

The search for new approaches to interdisciplinary collaboration is essential for navigating the complexities of system development in the face of increasing market demands. One conceivable approach to overcome this challenge is the adoption of a method based on heterogeneous models. This method allows various teams to create their standard models and conduct their usual analyses within their respective disciplines. Additionally, it offers the possibility of coupling these models to perform joint simulations (co-simulation), thus facilitating the analysis of the overall system behavior. The Functional Mock-up Interface (FMI) standard, developed by the Modelica Association, plays a crucial role in facilitating this integration. FMI provides a versatile framework for the easy exchange of models and simulation data between different tools and platforms.

In this work, we present a comprehensive approach to the taxonomy of co-simulation, using ontological notions to systematically categorize and define the various elements and interactions within co-simulation environments. This structured taxonomy serves as a basis for improving the clarity and interoperability of co-simulation processes. Furthermore, we focus on optimizing the orchestrator of OMSimulator, a platform designed for co-simulation based on the FMI standard. To achieve this, we employed an iterative method called the Aitken-Schwarz method. By integrating the Aitken-Schwarz method into the OMSimulator orchestrator, we were able to significantly improve the stability of numerical simulations, reducing errors and ensuring faster convergence of results.

These improvements enhance the co-simulation processes, thus providing a more robust and efficient solution for the development of complex systems in an interdisciplinary environment.

Keywords : Co-simulation, FMI, Taxonomy, OMSimulator, Aitken-Schwarz.

Table des matières

Table des figures

Liste des tableaux

Introduction générale

Contexte

Dans le domaine technologique, qui évolue rapidement, le développement de systèmes complexes exige souvent une collaboration entre plusieurs disciplines. Que ce soit en ingénierie aérospatiale, en conception automobile ou dans d'autres domaines, il est essentiel d'adopter des méthodes avancées pour naviguer à travers les complexités et interdépendances croissantes des composants de ces systèmes. Les approches traditionnelles, qui fonctionnent souvent dans des disciplines séparées, s'avèrent insuffisantes pour répondre aux exigences croissantes en matière d'efficacité, d'innovation et de performance. Il existe donc un besoin critique de nouvelles méthodologies collaboratives capables d'intégrer diverses expertises et de faciliter l'analyse complète des systèmes. L'utilisation de modèles hétérogènes et de techniques de co-simulation constitue une approche prometteuse pour relever ce défi. En permettant à différentes équipes de créer et d'analyser leurs modèles indépendamment tout en permettant des simulations intégrées, ces techniques offrent une voie vers un développement des systèmes plus robustes et plus précis. Le standard Functional Mock-up Interface (FMI), développé par l'association Modelica, apparaît comme un cadre essentiel dans le contexte de cette étude, fournissant une plateforme unifiée pour l'échange de modèles et de données entre différents outils de simulation.

Problématique

La co-simulation, outil crucial pour analyser et développer des systèmes complexes, est souvent abordée de façon spécifique par les différents communautés scientifiques et industrielles. Cette approche fragmentée mène à un manque de standardisation et de collaboration, rendant difficile l'établissement d'une base commune et solide pour le développement du domaine. Par conséquent, les équipes interdisciplinaires rencontrent des difficultés pour adopter cette méthode, ce qui limite son utilisation et freine l'innovation dans le secteur.

En outre, la majorité des plateformes de co-simulation existantes se trouvent soit à un stade de développement alpha, soit sont insuffisamment documentées, utilisant souvent des langages propriétaires. Cette situation pose de nombreuses contraintes à l'intégration et à l'incorporation d'algorithmes d'orchestration avancés. Les utilisateurs se retrouvent face à des défis significatifs pour implémenter et optimiser ces algorithmes dans des environnements de co-simulation non standardisés, ce qui limite l'efficacité et la fiabilité des

simulations numériques. Ainsi, il est impératif de développer des solutions robustes et bien documentées, basées sur des normes ouvertes comme le standard (FMI), afin de surmonter ces obstacles et de faciliter une adoption plus large et plus efficace des méthodes de co-simulation dans divers domaines.

Objectifs

Le premier objectif de cette étude est de développer une taxonomie simplifiée et basée sur des diagrammes pour la co-simulation. En utilisant une structure ontologique claire, nous allons catégoriser et définir systématiquement les différents éléments et interactions dans les environnements de co-simulation. Les diagrammes explicatifs rendront cette approche plus accessible et intuitive pour des équipes interdisciplinaires.

Le deuxième objectif est d'étudier les différentes plateformes de co-simulation existantes. Nous analyserons les caractéristiques, avantages et limitations des principales plateformes, en mettant en lumière les défis liés à l'utilisation de langages propriétaires. Cette étude permettra d'identifier les meilleures pratiques et les lacunes actuelles.

Enfin, le troisième objectif est de mettre en place un algorithme itératif d'orchestration pour la co-simulation. Cet algorithme sera utilisé pour renforcer la stabilité des simulations numériques, minimiser les erreurs et permettre une convergence accélérée des résultats.

Organisation du mémoire

Ce rapport est organisé en trois chapitres :

Le premier chapitre "**Présentation de l'organisme d'accueil et cadrage du projet**" présente l'entreprise d'accueil, la méthodologie, le planning, les contraintes liées au sujet, ainsi que le cadrage du projet.

Le deuxième chapitre "**Taxonomie de la co-simulation** " présente une catégorisation systématique et une définition des différents éléments et interactions au sein des environnements de co-simulation en utilisant des notions ontologiques.

Le troisième chapitre "**Conception et développement d'un orchestrateur de co-simulation** " explore une comparaison détaillée des différentes plateformes de co-simulation disponibles, mettant en lumière leurs caractéristiques, avantages et inconvénients. De plus, ce chapitre examine l'amélioration de l'algorithme d'orchestration, interprète les résultats obtenus et les compare avec ceux issus d'autres outils de co-simulation existants.

Chapitre 1

Présentation de l'organisme d'acceuil et cadrage du projet

1.1 Introduction

Ce chapitre présente Capgemini Engineering, une société de conseil spécialisée dans les solutions d'ingénierie, ainsi que l'équipe que j'ai intégrée et mon rôle dans le développement d'un système avancé d'aide à la conduite (ADAS) qui fonctionne efficacement dans des conditions météorologiques extrêmes.

Ce projet pose des défis uniques à cause de l'exclusivité de la technique et du nombre limité de publications disponibles sur le sujet. Cependant, il offre également des opportunités significatives de se plonger dans des domaines de pointe tels que la théorie du contrôle, l'apprentissage automatique et l'apprentissage profond.

Ce chapitre encadre le projet et fournit un contexte permettant au lecteur de comprendre ses objectifs, sa portée et ses résultats attendus, ainsi que le rôle de Capgemini Engineering dans son développement.

1.2 Présentation de l'organisme d'accueil

1.2.1 Capgemini Engineering

Capgemini Engineering est une filiale du groupe Capgemini qui rassemble les services d'ingénierie et de R&D de Capgemini Engineering, avec un chiffre d'affaires de 16 milliards d'euros. L'expertise de Capgemini dans le domaine de production numérique ainsi que sa connaissance sectorielle approfondie lui permettent d'accompagner les entreprises dans leur transformation vers la 4.0 en conjuguant les mondes physique et numérique. Avec plus de 52 000 ingénieurs et scientifiques dans plus de 30 pays , Capgemini Engineering intervient dans des secteurs variés tels que l'aéronautique, l'automobile, les sciences de la vie ou encore l'énergie. Le groupe Capgemini, quant à lui, est un leader mondial responsable et multiculturel comptant près de 270 000 collaborateurs dans une cinquantaine de

pays. Grâce à son expertise et son expérience, il répond à l'ensemble des besoins de ses clients, de la stratégie à la gestion des opérations, en utilisant les dernières innovations technologiques.

Fig. 1.1 : Logo de la société Capgemini Engineering

En tant que partenaire stratégique, Capgemini Engineering joue un rôle essentiel dans l'accompagnement complet des projets de ses clients à travers divers secteurs industriels. L'entreprise s'engage à fournir un niveau de service constant et de qualité, en offrant une expertise approfondie dans chaque domaine. Afin de répondre aux besoins spécifiques de ses clients, Capgemini Engineering adopte une approche à la fois mondiale et locale. En tant que groupe d'envergure mondiale, il est en mesure de proposer des solutions globales et innovantes qui répondent aux défis de l'industrie à grande échelle. Cependant, il ne perd pas de vue l'importance d'une présence locale et d'une compréhension approfondie des marchés spécifiques. Cela permet à Capgemini Engineering d'offrir un accompagnement personnalisé et adapté aux exigences particulières de chaque marché. Les clients et partenaires stratégiques de Capgemini Engineering jouent un rôle clé dans la réussite de l'entreprise. En collaborant étroitement avec ces acteurs majeurs de l'industrie, elle favorise l'innovation, la co-création et la recherche de solutions technologiques avancées. Grâce à ces partenariats solides, l'entreprise est en mesure d'anticiper les tendances du marché et de proposer des solutions différenciées qui répondent aux besoins changeants de ses clients.

1.3 Le projet RT-SIM

L'équipe RT-SIM est composé de cinq docteurs et quatre stagiaires (moi y compris). Le projet de recherche interne, RT-SIM, a pour but de développer une plateforme de simulation générique, distribué et temps réel.

1.3.1 Contexte

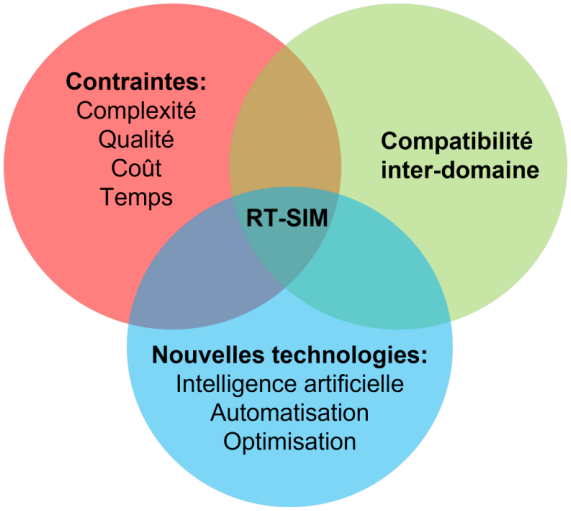
De nos jours, le développement d'un nouveau produit dans le secteur des transports, de l'industrie, de l'énergie et bien d'autres est devenu complexe. On souhaite améliorer la qualité d'un produit en y intégrant de nouvelles technologies, tout en diminuant le coût et le temps de développement. C'est ainsi que ces dernières années, il y eut un intérêt croissant pour la simulation. En effet, les simulateurs permettent d'intervenir sur tout le cycle en V d'un produit et permettent ainsi d'intégrer de nouvelles technologies tout en diminuant les coûts et le temps de développement. Les simulateurs devenant de plus en plus réalistes, il devient de moins en moins nécessaire de développer des prototypes. C'est ainsi qu'est né le projet RT-SIM qui souhaitait prendre en compte les nouvelles technologies dans le développement de sa plateforme de simulation, les contraintes imposés

dans les secteurs simulés et la capacité d'être compatible dans tous les domaines afin de pouvoir rendre un simulateur le plus réaliste possible.

1.3.2 objectif

L'objectif du projet RT-SIM est d'être :

1. Générique afin de faire coexister plusieurs secteurs d'activité.
2. Interopérable avec différents standards (ED-247 et FMI pour l'aéronautique, SMP2 pour le spatial, etc.), protocoles de bus de communication (CAN, ED-247, AFDX, etc.) et techniques de modélisation (discret et continu).
3. Distribué avec diverses ressources informatiques connectées par Ethernet.



1.4 Conclusion

Pour conclure, dans ce chapitre nous avons présenté l'organisme d'accueil **Capgemini Engineering** où ce projet a été réalisé, son historique, son organigramme et l'équipe de travail. En utilisant des différents outils, nous avons mis en évidence le cadre du projet, les objectifs visés, la démarche de travail et la planification de ce présent projet.
Après, on a montré les limitations causant la problématique traitée dans ce projet, et en comparant les solutions physiques nous avons abouti à la solution physique adoptée dans ce projet.

Dans le chapitre suivant, nous allons traiter la partie théorique de RADAR, comprendre le fonctionnement, le traitement de signal nécessaire, et la nature des données récupérées à la fin.
Ainsi que la motivation autour l'utilisation de l'apprentissage profond, et une définition de l'architecture des réseaux de neurones (NN).

Chapitre 2

Taxonomie de la co-simulation

Introduction

Les systèmes d'ingénierie intègrent de multiples éléments dans un processus complexe, comprenant des composants physiques, des logiciels et leurs interactions. Cette complexité pose de nombreux défis pour la modélisation et pour la simulation de ces systèmes. L'une des façons de simuler des systèmes comprenant plusieurs composants consiste à modéliser et à simuler l'ensemble du système à l'aide d'un seul outil, ce que l'on appelle la simulation monolithique. L'autre approche consiste à simuler le système dans le cadre d'une co-simulation.

La co-simulation se définit comme le couplage d'au moins deux unités de simulation distinctes. Ces unités peuvent différer par l'outil de simulation employé, l'algorithme de résolution mis en œuvre, ou encore le pas de temps de résolution utilisé. Une unité de simulation, quant à elle, représente une entité logicielle exécutable responsable de la simulation d'une partie spécifique du système [1, 2]. Cette approche présente un potentiel considérable et a été exploitée dans divers domaines, notamment l'industrie automobile [3], la gestion de l'électricité [4], le chauffage, la ventilation et la climatisation (HVAC) [5], le domaine maritime [6], et la robotique [7].

Dans ce chapitre, on entreprend une révision de l'état de l'art de la co-simulation, en mettant l'accent sur le standard FMI. Notre contribution réside dans la proposition d'une nouvelle approche pour la taxonomie de la co-simulation, visant à décomposer et à simplifier l'analyse de l'état de l'art en détaillant ses fonctionnalités, les exigences et les contraintes de la co-simulation. Cette approche permet une meilleure compréhension des tenants et aboutissants de la co-simulation, ouvrant ainsi la voie à de nouvelles avancées dans ce domaine.

Dans la section 2.1, on effectue une présentation et une critique de l'existant en introduisant le standard FMI dans le cadre de la co-simulation, la section 2.2 présente le processus d'orchestration des scénarios de co-simulation et ses contraintes. La section 2.3 contient les résultats de l'étude taxonomique menée.

2.1 Le standard FMI et la co-simulation

2.1.1 FMI

FMI (Functional Mock-up Interface) [8], est un standard élaboré dans le but de créer une interface unifiée pour l'exécution de modèles de systèmes dynamiques, facilitant ainsi l'interopérabilité entre les outils de modélisation et de simulation. L'essence de ce standard réside dans la génération et l'échange de modèles conformes à la spécification FMI entre différents outils. Ces modèles sont, selon le standard, des FMUs (Functional Mock-up Units). On distingue deux modes d'utilisation principaux : l'échange de modèle (ME), où le modèle, dépourvu de solveur intégré, requiert la prise en charge de la résolution par l'utilisateur, et la co-simulation (CS), où les modèles sont exportés avec un solveur de l'outil d'import.

L'idée fondamentale derrière le FMI est de permettre aux utilisateurs de combiner des modèles provenant de divers environnements de développement et de simulation, sans être enchaînés par les restrictions d'un outil particulier. En d'autres termes, les FMUs encapsulent les modèles dans une structure standardisée, ce qui permet à ces modèles d'être facilement intégrés dans un environnement de co-simulation, où ils peuvent interagir de manière cohérente avec d'autres composants du système. Cette approche favorise la réutilisation des modèles, réduisant ainsi les efforts de développement et améliorant l'efficacité de la modélisation et de la simulation dans divers domaines d'application.

Dans le cas de la co-simulation, le standard décrit une interface discrète du modèle dynamique, par exemple, étant donnés l'état interne à l'instant t_n, les entrées u_n, le pas de communication de la co-simulation h, p les paramètres fournis, et y_{n+1} le vecteurs des sorties à l'instant $t_{n+1} = t_n + h$ peut être noté comme suit :

$$y_{n+1} = \psi_p(u_n, h) \tag{2.1}$$

L'évolution de l'état et du temps est latente, et n'est pas décrite par le standard. Ainsi, les événements sont traités en interne. En plus, l'accès à l'état n'est possible[1] qu'aux points de communication $n \cdot h$.

2.1.2 La co-simulation

Malgré l'intérêt grandissant pour les bénéfices et les défis scientifiques de la co-simulation, ainsi que la diversité des outils disponibles, il est à noter qu'aucune étude connue n'a encore entrepris une analyse comparative approfondie des différentes plateformes de co-simulation et de leurs limitations.

Dans l'article [9], Gomes et al. réalisent une revue de la littérature technique portant sur les divers algorithmes développés, en segmentant la co-simulation en trois grandes catégories :

1. DE : Co-simulation basée sur des événements discrets.

[1]Pour la version 2.0.x du standard, pourtant la version 3.x offre la possibilité d'accès à tout instant t

2. CT : Co-simulation basée sur le temps continu.

3. Hybride : Approche hybride de la co-simulation.

Ils présentent ainsi une classification des algorithmes et des cas d'utilisation de la co-simulation, ce qui permet d'échanger des solutions et de fournir une compréhension plus approfondie de ce domaine.

2.2 L'orchestration de la co-simulation et ses contraintes

2.2.1 Orchestration

La co-simulation offre la possibilité de simuler, dans un seul environnement, des systèmes conçus de manière indépendante par différents experts. Les unités de simulation, peuvent être des entités autonomes, fonctionnant potentiellement sur des machines distinctes. Pour les connecter, un orchestrateur est requis. Cet orchestrateur, également désigné sous le terme d'algorithme maître, assure le transfert des données de sortie vers les entrées, conformément à un scénario de co-simulation prédéfini, tout en supervisant la progression du temps simulé au sein de chaque unité de simulation. Les paramètres nécessaires pour garantir le bon déroulement de la co-simulation sont regroupés sous le nom de scénario de co-simulation. L'objectif principal de l'orchestrateur est de garantir que les simulations intégrées fonctionnent ensemble de manière stable et précise.

2.2.2 Contraintes d'orchestration

Dans [9], les contraintes sont subdivisées selon la segmentation présentée. On note dans le cas d'une *co-simulation basée sur des événements discrets* (DE) :

- **La causalité** : L'absence de violation de la causalité par chaque unité de simulation esclave est essentielle. Ainsi, tout orchestrateur doit veiller à maintenir cette causalité intacte lors de leur couplage.

- **Déterminisme et confluence** : La garantie de l'unicité de la trace de la co-simulation relève de la fonction select, comme décrit dans [9]. Une alternative à cette fonction est de s'assurer que toutes les combinaisons d'exécutions possibles conduisent toujours à la même trace de comportement, concept connu sous le nom de "confluence". Une unité de co-simulation conforme à la composition en termes de confluence l'est également en termes de déterminisme.

- **Structure dynamique** : Du point de vue des performances, suivre une séquence statique de dépendances peut s'avérer excessivement prudent. Avec le temps, cette chaîne de dépendances peut nécessiter des ajustements, entraînant un recours fréquent à la fonction "Rollback" pour rectifier la séquence à chaque nouvel événement, ce qui peut impacter négativement la performance de la co-simulation. Une

approche dynamique de la co-simulation permet aux chaînes de dépendances d'évoluer en fonction des changements dans le comportement des unités de simulation au fil du temps. [13]

On note dans le cas d'une *co-simulation basée sur le temps continu* (CT) :

- **Modularité et couplage** : Simplifier le scénario de la co-simulation est une contrainte importante dans le cadre de l'orchestration. En effet, la rigidité et la protection (Boîte noire) de chaque unité de simulation impliquent des complications lors du couplage de ces derniers. (Plusieurs exemples et approches sont détaillés dans [9]). La modularité se réfère à la capacité de diviser le scénario en blocs qui peuvent être réutilisés (légèrement modifiés) dans d'autres applications.
- **Boucles algébriques** : On distingue globalement deux types de boucles algébriques : celles qui n'impliquent que les entrées et celles qui s'étendent sur les variables d'état. Dans [14] Kübler et Schiehlen analysent la stabilité (zero-stability) de la co-simulation, et ils proposent deux méthodes : itérative[2] (résoudre les entrées inconnues pour chaque pas de simulation) ou bien éliminer les boucles algébriques en introduisant des filtres supplémentaires dans le modèle mathématique.
- **Initialisation** : Par définition, la condition initiale fait partie intégrante de l'unité de simulation. Cependant, dans le contexte d'une co-simulation, la simple présence d'une condition initiale pour chaque unité ne suffit pas. Il est crucial d'établir un point de départ commun et cohérent pour l'ensemble des unités afin d'assurer une co-simulation à la fois correcte et convergente. C'est ce que l'on nomme la co-initialisation.
- **Contrôle d'erreur** : Évaluer la précision d'une trace de co-simulation pose un défi majeur. Idéalement, la comparaison avec la trace réelle, obtenue par la solution analytique du système continu, permettrait de quantifier l'erreur et donc la précision. Cependant, dans la pratique, l'accès à la solution analytique est souvent impossible et la connaissance du modèle complet n'est pas toujours disponible. Cette absence de référence absolue rend la mesure de la précision d'une trace de co-simulation particulièrement complexe.

La plupart des modèles ne se limitent pas exclusivement à une catégorie parmi celles mentionnées précédemment. Ils se composent souvent d'unités à la fois continues et discrètes, ce qui les classe dans la catégorie des systèmes hybrides. Cependant, il n'existe pas de standard spécifiquement défini pour ces modèles hybrides. Néanmoins, des extensions du standard FMI sont proposées dans les références [24, 25].

[2] Une technique d'itération à point fixe qui utilise le retour en arrière (rollback) pour répéter l'étape de co-simulation avec des entrées corrigées est appelée itération dynamique, itération 'waveform relaxation' et couplage fort ou en oignon.

On note que les limitations de cette approche sont :

- **Gestion du pas de communication :** En cas de gestion des événements discrets dans l'approche hybride, les événements sortants des unités de simulations peuvent être ignorés. Pour cela plusieurs approches ont été conçues telle que la gestion predictive du pas de communication [26].
- **Localisation et gestion des discontinuités :** La localisation du moment exact où un signal continu franchit un seuil est un problème bien connu, et intimement lié à l'estimation de l'avance temporelle pour prédire le pas de communication.
- **Stabilité et tolérance :** Il existe toujours une possibilité qu'une succession d'événements puisse causer l'instabilité du système. Des méthodes d'analyse et d'identification de ces points d'instabilités permettent d'améliorer la stabilité et rester dans les tolérances spécifiées.

2.3 Taxonomie de la co-simulation

Dans [9], une structure arborescente est présentée, segmentant les fonctionnalités de la co-simulation, tant au niveau des plateformes que des unités. En revanche, dans l'article [27], les auteurs proposent une taxonomie basée sur une carte mentale (Mind-Map), détaillant les options et les aspects à considérer du point de vue de l'utilisateur.

L'un des objectifs principal des taxonomies est de structurer une ontologie, de faciliter la compréhension humaine et d'aider à l'identification et la résolution des problèmes. Toutefois, la simplicité des relations taxonomiques a parfois conduit à des interprétations incomplètes, soulignant ainsi la nécessité de recourir à des techniques d'analyse plus sophistiquées. Dans ce travail, on a renforcé ces liens en utilisant des notions ontologiques [28] adaptées aux besoins spécifiques du système.

Il est également crucial de comprendre que la pensée architecturale diffère en partie de la pensée analytique classique. Raisonner en tant qu'architecte implique moins de se concentrer sur les détails opérationnels du système que sur l'identification des grands invariants structuraux. Cela permet ensuite au système de trouver son propre équilibre tout en respectant le cadre architectural qui lui a été donné, et qu'il doit préserver structurellement, comme le souligne l'article [29].

Les propriétés qu'on utilise dans ce travail se décomposent comme suit :

- Dépendances : Il existe une relation de dépendance entre deux unités lorsque l'existence et le bon fonctionnement de l'une dépendent de l'autre.
- Contraintes : Ces relations génèrent des conflits, que ce soit au niveau du développement ou de l'utilisation des systèmes. (Représentées par 'C')
- Liaisons d'hérédité : Elles se produisent lorsqu'un objet transmet un ensemble de ses propriétés à un autre. (Représentées par 'H')
- Approches : Ces éléments représentent les propositions destinées à l'utilisateur comme des propriétés.

- Objets : Ce sont des entités regroupant diverses fonctionnalités et exigences. (Représentés sous forme de tables)

2.3.1 Méthodologie et résultats

Après avoir passé en revue plusieurs articles, notamment ceux référencés [9–14], ainsi que les plateformes de co-simulation [15–18, 20, 22, 23], on a adopté une approche en deux étapes. Dans un premier temps, cette approche permet de mettre en évidence les exigences et les catégories fondamentales de la co-simulation. Ensuite, elle explore les contraintes, les dépendances et les différentes approches envisagées par les acteurs impliqués dans le processus de co-simulation.

Cette méthode a permis de mieux comprendre la complexité de la co-simulation en identifiant clairement les exigences et les contraintes inhérentes à cette pratique. En décomposant le processus en étapes distinctes, on a pu cerner les défis spécifiques et les besoins des utilisateurs finaux. Cette approche a également permis d'identifier les points moins traités dans la littérature.

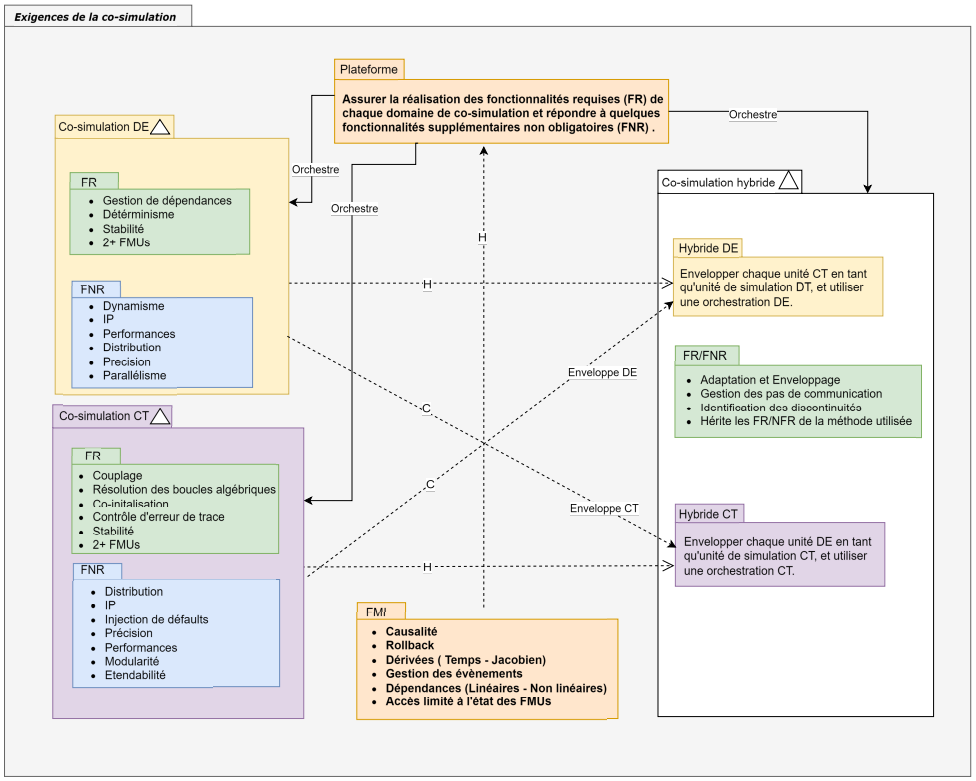


FIG. 2.1 : Diagramme d'exigences internes de la co-simulation

La figure 2.1 offre une classification de la co-simulation, mettant en lumière les différentes propriétés requises ou optionnelles (FR/NFR) offertes par les divers intervenants de ce processus. Elle illustre les relations entre ces entités, telles que la liaison d'hérédité entre la plateforme et le standard FMI, offrant ainsi une vision holistique des interactions dans le domaine de la co-simulation.

D'autre part, la figure 2.2 dépeint d'une manière fonctionnelle les exigences décrites

dans la figure 2.1. En fournissant une représentation graphique du cheminement des exigences à travers les différentes entités de la co-simulation, cette figure offre une vue détaillée de la dynamique fonctionnelle de ce processus complexe.

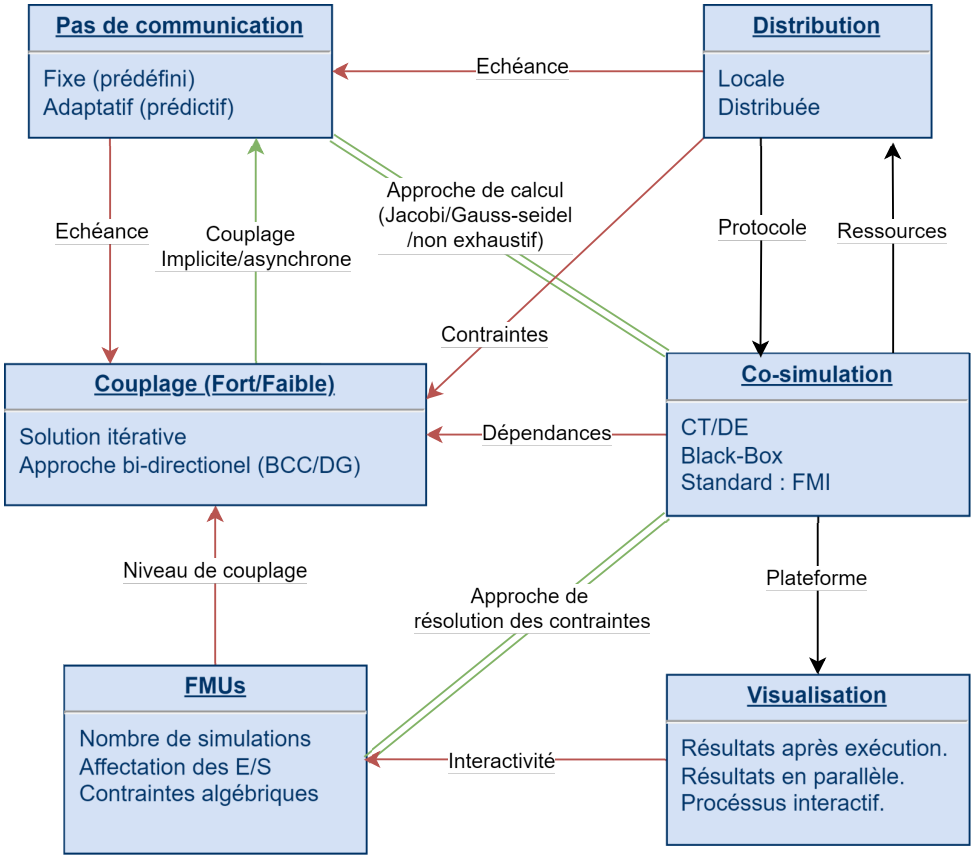


Fig. 2.2 : Diagramme d'exigences fonctionnelles de la co-simulation

2.4 Conclusion et perspectives

Ce chapitre met en lumière les défis intrigants du domaine de la co-simulation. L'analyse commence par explorer séparément les deux principales approches : la co-simulation basée sur le temps continu et celle sur les événements discrets, avant d'examiner les difficultés associées à leur intégration. Une taxonomie basée sur des principes ontologiques et structurée en deux niveaux est ensuite introduite. La figure 2.1 illustre les interactions et les rôles des différentes entités impliquées, offrant un aperçu complet des aspects clés du processus, et servant également de modèle conceptuel pour le développement d'outils de co-simulation. Enfin, la figure 2.2 détaille les exigences de base et les approches correspondantes pour chaque entité, enrichissant ainsi notre compréhension de la structure fonctionnelle du sujet.

Au cours de cette étude, on a passé en revue plusieurs plateformes de co-simulation majeures, telles que Maestro2 [15], OMSimulator [16] et Daccosim ng [17]. Une analyse comparative de ces plateformes, basée sur les fonctionnalités clés identifiées dans cette étude, fera l'objet du prochain chapitre.

Chapitre 3

Plateforme de co-simulation et algorithme d'orchestration

Introduction

Les plateformes de co-simulation nécessitent des algorithmes d'orchestration (OA) pour coordonner l'exécution des FMUs dans un scénario. L'OA définie comment les données sont échangées entre les FMUs dans le scénario et permet également d'influencer leur évolution d'état durant la co-simulation. Bien qu'il ne fasse pas partie de la norme FMI, l'OA est une caractéristique essentielle d'une plateforme de co-simulation. Des études ont démontré qu'il a un impact considérable sur l'exactitude des résultats de la co-simulation [9].

Malheureusement, il n'y a pas de solution open source pour la co-simulation qui combine flexibilité, facilité d'usage et performances. Ce manque oblige ceux qui veulent personnaliser leur processus ou expérimenter avec de nouveaux algorithmes à construire leur propre système depuis le début ou reprendre un énorme travail de choix et compréhension des plateformes existantes (en prenant en compte les problèmes de licence). Ceci est particulièrement chronophage pour les grands projets, car il faut coder manuellement toutes les interactions entre les unités de simulation, prolongeant ainsi la phase préparatoire avant même de commencer à optimiser la co-simulation.

Dans ce chapitre, on va présenter une comparaison des différentes plateformes open source, où on va spécifier les points forts et faibles de chacune et argumenter le choix de notre plateforme de base (OMSimulator [16]). D'autre part, on présente le processus de développement d'un orchestrateur basé sur la méthode d'Aitken-Schwarz [32, 33].

3.1 Plateformes de co-simulation

Le standard FMI joue un rôle essentiel dans les plateformes de co-simulation, en facilitant l'intégration et l'interopérabilité des modèles entre différents outils. Des plateformes notables comme Dymola, OpenModelica, Simulink, Maestro2 et Open Simulation Platform (OSP) utilisent largement la norme FMI pour améliorer leurs capacités de si-

mulation. Dymola [19] permet un échange de modèles et une co-simulation efficaces grâce à l'interface FMI, et prend en charge divers domaines. OpenModelica a développé OMSimulator [16], qui peut servir comme une plateforme de co-simulation indépendante à base du standard FMI. Simulink [20], de MATLAB, prend en charge le standard FMI à la fois en tant qu'importateur et exportateur, s'intégrant ainsi de manière transparente dans les flux de travail multi-outils. Maestro2 [15], offre de solides fonctions de co-simulation à l'aide de FMI, visant à simplifier l'intégration de systèmes multidisciplinaires complexes. DACCOSIM NG [17], la dernière version de la plateforme de co-simulation Daccosim, offre plusieurs fonctionnalités notamment la distribution de la co-simulation. PyFMI [18] est un package Python qui offre une interface pour travailler avec des modèles basés sur le standard FMI. Enfin, OSP [21] est une plateforme de co-simulation axée sur l'industrie maritime.

3.1.1 Limites et choix

La création d'une plateforme de co-simulation à partir de zéro est souvent inefficace et coûteuse en ressources, étant donnée la complexité de l'intégration et de la maintenance requise. Opter pour une base existante permet non seulement de réduire significativement le temps et les efforts de développement, mais aussi d'assurer une compatibilité et une robustesse initiale. La sélection d'une plateforme appropriée s'oriente selon des critères —tels que la personnalisation, les performances, SSP[1], et la documentation— qui sont détaillés dans la figure 3.1. Cette approche stratégique favorise une réalisation plus efficace des objectifs de l'étude, concentrant les efforts sur la personnalisation plutôt que sur la construction initiale.

Tab. 3.1 : Comparaison basée sur divers critères

Critère	OMSimulator	Maestro2	OSP	pyFMI	Daccosim NG
Personnalisation	✓✓✗	✓✓✓	✓✗✗	✓✓✗	✓✗✗
Performances	✓✓✓	✓✓✓	✓✗✗	✓✓✗	✓✓✗
Documentation	✓✓✓	✓✗✗	✗✗✗	✓✓✗	✓✗✗
Facilité d'utilisation	✓✓✓	✓✗✗	✓✓✗	✓✓✓	✓✓✗
Support de SSP	✓✓✓	✓✓✗	✓✓✗	✓✓✗	✓✗✗
Modularité	✓✓✗	✓✓✓	✓✗✗	✗✗✗	✓✓✗
Distribution	✗✗✗	✗✗✗	✗✗✗	✓✓✗	✓✓✓
Langage	C++	Java	C++	Python	C++

J'ai opté pour OMSimulator pour plusieurs raisons convaincantes. Tout d'abord, il bénéficie d'une documentation riche et accessible, ce qui facilite grandement l'apprentissage et l'utilisation de l'outil. Sa structure et sa conception soignées garantissent également une expérience utilisateur fluide et aisée. De plus, OMSimulator est un outil déjà bien développé, offrant une base solide pour tout projet de co-simulation. Un autre point fort est

[1]Le standard SSP (System Structure and Parameterization) est conçu pour améliorer l'interopérabilité entre différents outils de simulation en permettant une description structurée des systèmes et de leurs paramètres.

l'activité continue de son équipe de développement sur GitHub, ce qui témoigne de leur engagement à améliorer constamment l'outil et à soutenir sa communauté d'utilisateurs. Cette combinaison de facteurs fait d'OMSimulator le choix idéal pour ce projet.

3.1.2 OMSimulator

OMSimulator [16] est développé en tant que bibliothèque de simulation autonome en open source, dotée d'une interface utilisateur complète en C. Son intégration dans l'éditeur graphique OpenModelica OMEdit illustre l'utilisation de l'interface C-API pour créer une expérience utilisateur graphique et intuitive. En outre, OMSimulator propose une interface en ligne de commande (CLI), des interfaces de script pour Python et Lua, permettant son intégration dans divers outils tiers et applications spécialisées, tels que des simulateurs de vol.

Dans OMS, les modèles composites sont structurés sous forme d'un arbre hiérarchique composé de blocs de construction spécifiques. Le nœud racine de cet arbre peut être un système TLM, un système faiblement couplé (système WC) ou un système fortement couplé (système SC), chacun se distinguant par la manière dont les connexions sont gérées :

- Les systèmes **TLM** contiennent des connexions TLM, qui peuvent être considérées comme des connexions retardées motivées par la physique.
- Les systèmes faiblement couplés **Weakly-Coupled** sont utilisés pour la co-simulation. Toutes les unités de simulation fonctionnent de manière indépendante et sont synchronisées par un algorithme maître à certains moments de la communication.
- Les systèmes fortement couplés **Strongly-Coupled** sont utilisés pour regrouper les FMUs-ME dans une unité de co-simulation. Ils partagent un solveur commun et utilisent un schéma de communication continu.

On a étudié la documentation générée à travers Doxygen[2] et le source code fournit[3] afin de créer une documentation plus ciblée qui servira de guide pour les futures modifications par l'équipe RTSIM. Cette documentation spécifique est présentée dans l'annexe 3.5. Il est important de noter que notre étude se concentre particulièrement sur l'approche des systèmes faiblement couplés (WC) et que tous les FMUs utilisés pour les démonstrations dans cette recherche sont des FMUs-CS.

3.1.3 Interface d'utilisation OMS

Dans notre contexte, on se sert de la version lignes des commandes, ainsi que l'interface python pour la rédaction des scénarios de co-simulation.

Dans la configuration d'OMSimulatorPython, le fichier 'capi.py' joue un rôle essentiel puisqu'il utilise 'ctypes', une bibliothèque Python pour appeler des fonctions C depuis

[2] https://www.doxygen.nl/
[3] https://github.com/OpenModelica/OMSimulator

Python. Cela permet aux scripts Python d'accéder directement et d'utiliser les fonctionnalités de simulation sous-jacentes fournies par OMSimulator. En utilisant 'ctypes' pour établir des liaisons avec des fonctions C, 'capi.py' convertit efficacement ces fonctions en fonctions que l'on peut invoquer en Python. Autour de ce fichier central, d'autres modules Python tels que 'Model.py', 'System.py', et 'Scope.py' s'appuient sur ces liaisons pour offrir une API plus abstraite, permettant aux utilisateurs de gérer les scénarios de co-simulation plus facilement. Cette architecture non seulement simplifie l'intégration de tâches de simulation complexes dans les scripts Python, mais maintient également les avantages de performance de l'implémentation C native, comblant ainsi le fossé entre la facilité de scriptage de haut niveau et la puissance computationnelle de bas niveau.

Dans la figure A.1, on présente un script Python pour lancer une co-simulation d'un amplificateur différentiel divisé en deux FMUs comme présenté dans la figure suivante :

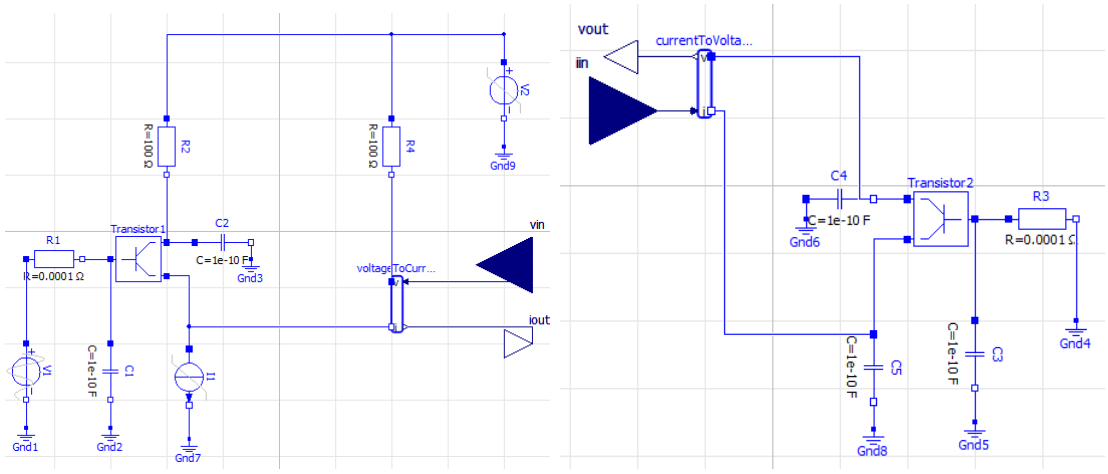


Fig. 3.1 : Modèles des FMUs utilisés dans la co-simulation de l'amplificateur différentiel

3.2 Génération des FMUs

Transformer un modèle ou un ensemble d'équations en un FMU , qui compile essentiellement ces éléments en un ensemble de code C et de bibliothèques, présente un ensemble unique de défis et d'opportunités. Cette transformation implique l'encapsulation du comportement dynamique du modèle dans une unité compilée autonome qui interagit avec divers environnements de simulation via le standard FMI. Cependant, la compilation multi-plateforme représente un problème notable. Assurer le fonctionnement sans faille d'un FMU sur différents systèmes d'exploitation (Windows, Linux, MacOS) peut être complexe en raison des différences entre les compilateurs, les bibliothèques et les architectures systèmes.

On présente quelques outils de génération des FMUs testés dans cette étude :

- **SIMULINK** : On utilise le Simulink Coder en combinaison avec le FMI Kit pour générer des FMUs à partir des modèles Simulink. Cette méthode permet d'exporter les modèles sous forme de FMUs, supportant la co-simulation ou l'échange de

modèles. Le processus inclut la configuration du modèle Simulink pour l'exportation, la sélection des réglages appropriés du solveur, puis l'utilisation de la fonction d'exportation du FMI Kit pour finaliser l'exportation.

- **OpenModelica** : OpenModelica offre un support intégré pour l'exportation de modèles sous forme de FMUs. On charge simplement le modèle dans OpenModelica et on utilise la commande exportToFMU(). Cette commande permet de spécifier différentes options, telles que la compatibilité de version (FMI 1.0 ou 2.0), et le choix entre l'exportation pour la co-simulation ou l'échange de modèles. On peut également utiliser le drapeau supplémentaire "-d=fmuExperimental". Ce drapeau permet d'activer des fonctionnalités expérimentales spécifiques qui ne sont pas encore standard dans la version stable de OpenModelica tel que : fmi2GetSpecificDerivatives, canGetSetFMUState, canSerializeFMUstate.

- **Source Code FMU (pythonfmu)** : Ces dernières années, plusieurs cadres logiciels open-source ont été développés pour l'exportation des FMUs à partir du code source. CPPFMU, développé par SINTEF Ocean dans le cadre du projet ViProMa, propose une interface C++ de haut niveau avec des fonctionnalités telles que les exceptions et la gestion automatique de la mémoire pour écrire du code maîtres/esclaves conforme à FMI. Cependant, il ne prend pas en charge la génération du fichier 'modelDescription.xml' ni le conditionnement des FMUs. FMUSDK, fourni par QTronic, est un SDK basé sur C permettant l'utilisation des FMUs pour l'échange de modèles et la co-simulation, et qui supporte jusqu'à la version 2.0 de FMI, mais il manque également la génération automatique du modelDescription.xml. JavaFMI, développé par l'institut SIANI et financé par l'EIFER, et FMI4j, développé en Kotlin et sous licence MIT, prennent tous deux en charge l'importation et l'exportation de FMUs sur la JVM, FMI4j utilise CPPFMU pour la mise en œuvre des fonctions FMI et se concentre sur la performance grâce à l'intégration JNI. JavaFMI emploie une approche plus impérative utilisant le passage de messages, tandis que FMI4j utilise un style déclaratif avec des annotations, conduisant à des différences dans la performance et l'interaction utilisateur dans la définition du modèle.

Tool	Target language	Target platform	FMI version
JavaFMI	JVM	Win, Linux	2.0
FMI4j	JVM	Win, Linux	2.0
CPPFMU	C++	Win, Linux	1.0 & 2.0
FMUSDK	C	Win, Linux, OSX	1.0 & 2.0
Pythonfmu	python	Win, Linux	1.0 & 2.0

Tab. 3.2 : Comparaison des différents outils d'export des FMUs

PythonFMU [4] [30] est une plateforme logicielle sous licence MIT qui facilite l'empaquetage de code Python "3.x" en FMUs de co-simulation. Développé en collaboration entre NTNU et Safran Tech, il est accessible via pip ou conda. Bien qu'il fonctionne immédiatement sur les systèmes Windows et Linux 64 bits, PythonFMU

[4] https://github.com/NTNU-IHB/PythonFMU

requiert une distribution Python compatible déjà installée sur le système cible, ainsi que toute bibliothèque tierce nécessaire. On présente dans A.2, notre script python pour créer un FMU correspondant à une source de courant.

- **uniFmu** [5] [31] : Universal Functional Mock-up Unit est un outil qui permet l'implémentation des FMUs dans n'importe quel langage de programmation. Ce framework fournit des binaires pré-compilés pour Windows, Linux et macOS, évitant ainsi les complications liées à la cross-compilation et à la configuration de chaînes d'outils complexes. Il intègre également un mécanisme d'extension facile à utiliser, permettant la prise en charge de divers langages de programmation. De plus, une interface en ligne de commande est disponible pour générer rapidement des FMUs à l'aide d'une seule commande.

3.3 Orchestration de la co-simulation

L'algorithme orchestrateur est chargé de gérer l'échange de données et la synchronisation entre ses composants, assurant ainsi que la simulation globale respecte les contraintes temporelles et les exigences de précision préétablies. Il prend en charge des tâches telles que la gestion des pas de temps, le contrôle des erreurs et la séquence d'exécution des différentes unités de simulation. En orchestrant la manière et le moment où ces FMUs échangent des données, l'algorithme maître vise à assurer que les simulations restent cohérentes et pertinentes.

3.3.1 Orchestrateurs d'OMS

Comme mentionné dans A.2, on s'intéresse à la classe SystemWC.cpp. On présente dans cette section les deux algorithmes orchestrateurs d'OMS.

Solveur	Type	Description
oms_solver_wc_ma	wc-system	Algorithme maître par défaut à pas fixe
oms_solver_wc_mav	wc-system	Algorithme maître à pas adaptatif
oms_solver_wc_mav2	wc-system	Algorithme maître à pas adaptatif (double pas)

Tab. 3.3 : Description des orchestrateurs utilisés dans les systèmes wc

On note d'abord des points en commun entre ces différents orchestrateurs :

- Méthode **stepUntil** : Responsable de l'avancement du temps des différents FMUs présents dans le scénario.
- Méthode **getInputAndOutput** : Cette méthode traite les connexions entre les composants au sein d'un graphe dirigé pour extraire et gérer les valeurs d'entrée et de sortie pour les FMUs capables de gérer leur propre état. Elle parcourt les connexions triées, en excluant les boucles algébriques, et remplit les vecteurs avec les valeurs des signaux réels des composants FMU connectés.

[5] https://github.com/INTO-CPS-Association/unifmu/tree/

- Méthode **SolveAlgLoop** : Cette méthode offre à l'utilisateur le choix entre deux solveurs pour résoudre les boucles algébriques, en fonction de la complexité du problème. KINSOL de SUNDIALS est idéal pour les systèmes d'équations non linéaires complexes grâce à ses méthodes robustes comme celles de Newton, adaptées aux modèles de grande envergure nécessitant des calculs précis de Jacobien. En revanche, FixedPointIteration (Algorithme de Tarjan) est plus adapté pour les systèmes moins complexes, utilisant une itération de point fixe qui permet une résolution rapide et efficace pour des problèmes d'ingénierie standard.

Dans A.5, on décrit l'algorithme qui illustre le fonctionnement de l'algorithme maître utilisant une taille de pas adaptative. Le principe repose sur la gestion de l'erreur, définie comme la différence entre les interfaces de deux états successifs, où le système est régulé en fonction de la magnitude de l'erreur. Cette dernière est indicative du dynamisme du système. On note trois points faibles de cet algorithme :

- **Contrôle de la Taille de Pas** : Si les tailles de pas maximum et minimum sont mal ajustées, cela pourrait empêcher la convergence de l'algorithme ou causer de l'instabilité par des pas trop grands.
- **Gestion des Erreurs** : La méthode calcule l'erreur et utilise un facteur de sécurité pour décider des rollbacks. Une mauvaise calibration de ces seuils peut entraîner des rollbacks inutiles ou insuffisants, affectant l'efficacité et la précision de la simulation.
- **Stabilité numérique et convergence** : L'algorithme ajuste dynamiquement les tailles de pas en fonction de la réponse du système, ce qui présente un risque d'instabilité numérique si les ajustements de pas ne sont pas en accord avec les propriétés numériques du système. Cela pourrait entraîner des solutions qui ne convergent pas ou un comportement divergent, particulièrement dans les systèmes sensibles ou très dynamiques.

Dans A.5, on décrit l'algorithme utilisé par oms_solver_wc_ma, qui implémente une méthode de pas fixe pour la simulation. Cet algorithme intègre également la sauvegarde des états des composants pour permettre des rollbacks en cas d'erreurs durant les étapes de simulation. Deux aspects critiques de cet algorithme sont notés :

- **Adaptabilité aux Modèles Non Linéaires et Complexes** : Les simulations qui impliquent des modèles non linéaires ou ayant des comportements complexes posent des défis particuliers pour les méthodes à pas fixe.
- **Critères de Convergence** : Pour garantir que l'algorithme aboutisse à une solution correcte et stable, il est important d'établir des critères de convergence bien définis et bien personnalisé. Dans le cas d'une méthode à pas fixe, où le seul degré de liberté réside dans la modification des dérivées des signaux d'entrée, cette approche peut s'avérer limitée face à la dynamique de certains systèmes.

Cela met en lumière l'importance d'adopter une méthode qui marie l'adaptabilité du pas de communication, l'optimisation des critères de convergence, et une gestion d'erreur solide, tout en maintenant la complexité des calculs à un niveau minimal.

3.4 Orchestrateur Aitken-Schwarz

Dans les sections précédentes, nous avons abordé plusieurs problèmes et défis liés à la co-simulation, incluant l'instabilité causée par le pas de communication et les mécanismes d'estimation et de contrôle d'erreurs. Ces difficultés sont largement reconnues et étudiées dans la littérature scientifique. Dans [34], une méthode itérative est proposée, qui assure la cohérence des interfaces tout en éliminant les discontinuités à chaque macro-étape. Cette méthode a été comparée à d'autres approches bien établies, telles que la co-simulation non itérative type Jacobi, la co-simulation itérative avec conservation de l'ordre zéro [14], et un algorithme non itératif amélioré pour le lissage des variables [35].

Dans notre travail, nous considérons un algorithme de couplage en point fixe basé sur la technique de décomposition du domaine de Schwarz dans lequel nous avons utilisé un Schwarz additif (ASM) [32]. Ces méthodes itératives peuvent être convergentes ou divergentes en fonction du partitionnement du domaine et des conditions aux limites. En outre, nous avons utilisé la propriété de convergence ou de divergence purement linéaire (c'est-à-dire que l'opérateur d'erreur de la méthode ne dépend pas du nombre d'itérations) pour accélérer la méthode itérative vers la vraie solution avec la technique d'accélération de la convergence d'Aitken, même avec une méthode divergente [33].

Il est important de noter que, dans notre contexte, l'accélération est spécifiquement appliquée à la résolution de l'interface. De plus, ces méthodes sont non intrusives pour les FMUs, ce qui les rend particulièrement adaptées à la co-simulation de systèmes de type boîte noire (FMU-CS).

3.4.1 Implémentation de la méthode Aitken-Schwarz

On note $z(t_n)$ le vecteur contenant les différents variables de l'interface du modèle au pas de communication t_n, où $z(t_n) \in \mathbb{R}^N$. Le vecteur $z(t_{n+1})$ l'interface après un pas de communication (c'est-à-dire après un appel de la fonction **stepUntil** $\mathcal{S}$).

$$z(t_{n+1}) = \mathcal{S}(z(t_n)) \tag{3.1}$$

Le principe de cette méthode consiste à réaliser, au cours d'un unique pas de communication, plusieurs itérations (c'est-à-dire de multiples pas de simulation sans progression temporelle), jusqu'à ce que le critère de convergence soit atteint ou qu'une limite maximale d'itérations soit dépassée.

$$Z^{k+1} = \mathcal{S}(Z^k) \tag{3.2}$$

où k est l'indice d'itérations de Schwarz. En outre, comme montré dans [33] :

$$Z^{k+1} - Z^0 = \mathbb{P}(Z^k - Z^0) \tag{3.3}$$

on peut extraire de l'équation (2.3) la forme généralisé de l'accélération d'aitken :

$$Z^{\infty} = (Id_N - \mathbb{P})^{-1}(Z^1 - \mathbb{P}Z^0) \tag{3.4}$$

En définissant l'erreur entre les valeurs à l'interface pour les itérations (k+1) et (k) de la manière suivante : $e^k = (Z^{k+1} - Z^k)$, il est possible de calculer algébriquement l'opérateur

$\mathbb{P} \in \mathbb{R}^{N \cdot N}$ à partir de l'équation (2.3) après $N+1$ itérations. Cette démarche est réalisable si la matrice $[e^N, \ldots, e^1]$ est inversible, et s'effectue de la manière suivante :

$$\mathbb{P} = [e^{N+1}, \ldots, e^2] \cdot [e^N, \cdots, e^1]^{-1} \tag{3.5}$$

Cet opérateur d'erreur doit uniquement être calculé lors de la première étape. Toutefois, si la topologie évolue ou en présence de non-linéarités, il sera nécessaire de recalculer l'opérateur. [33]

La méthode a été implémenté dabs un méta-modèle de co-simulation et testé avec un circuit RL. Le code correspondant est présenté dans A.1.

3.4.2 Implémentation dans OMSimulator

L'algorithme qui résume l'orchestrateur Aitken-Schwarz est présenté dans la figure ci-dessous :

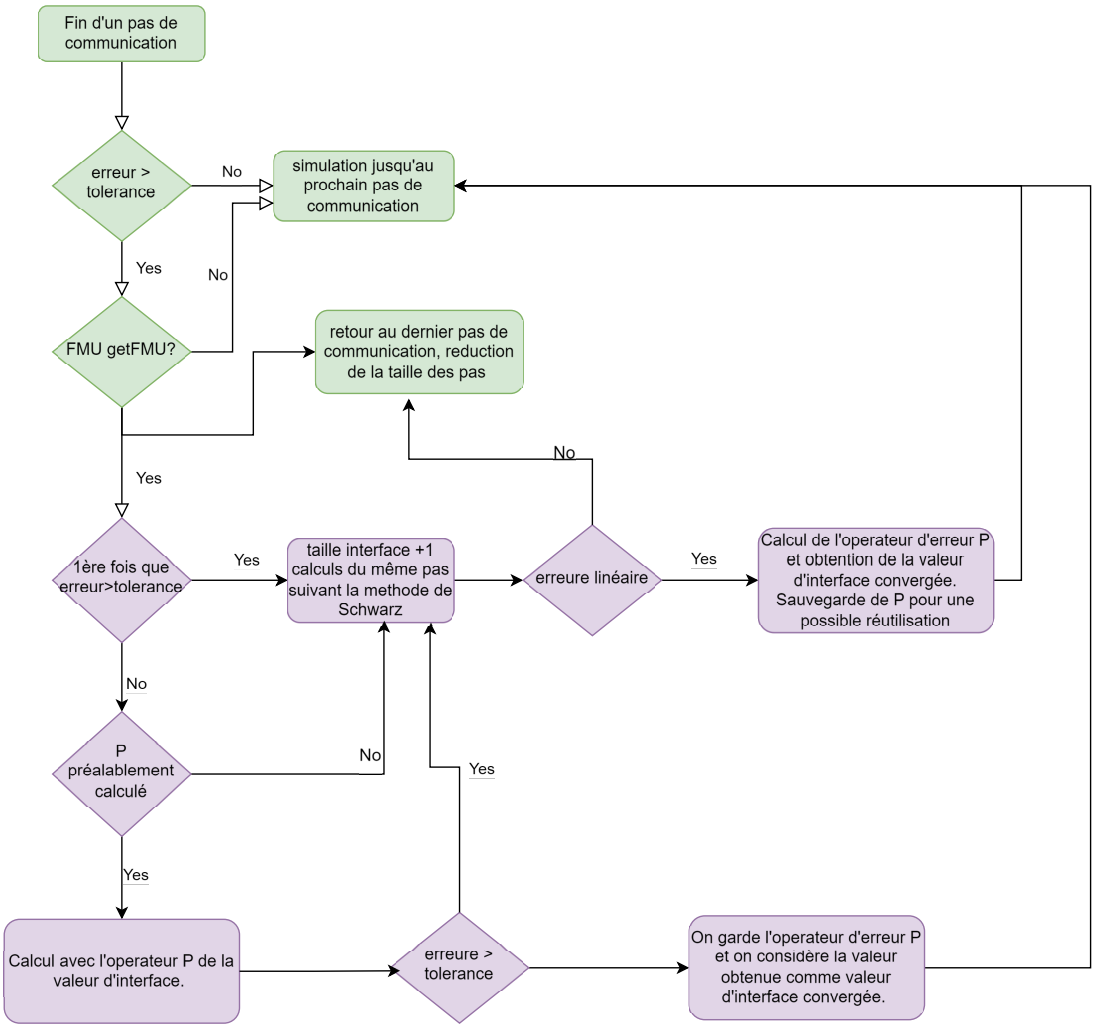


FIG. 3.2 : Algorithme de l'orchestrateur Aitken-Schwarz

L'algorithme repose sur le principe du rollback et requiert donc des FMUs compatibles avec cette fonctionnalité.

3.4.3 Tests et résultats

Pour générer des scénarios de co-simulation, on a découpé des modèles OpenModelica, avant de les exporter sous forme de FMUs. On effectue dans cette section, des tests et comparaison entre trois algorithmes orchestrateurs : - oms_solver_wc_ma (Rollback et modification de dérivée) - oms_solver_wc_AS (Rollback et Aitken-Schwarz) - Orchestrateur simple avec tolérance relative.

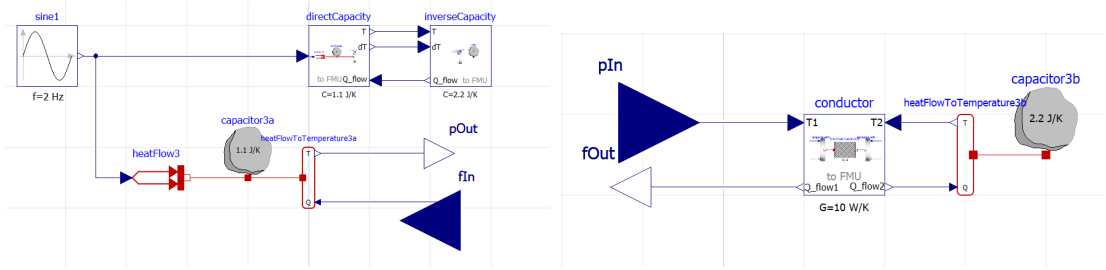


FIG. 3.3 : Modèles des FMUs utilisés dans la co-simulation de l'amplificateur différentiel

Ce modèle est censé pour une co-simulation d'un système thermique avec une entrée sinusoïdale influençant des propriétés thermiques telles que la capacité et la conductance. On a découpé le modèle pour créer un point d'échange (pOut (Température T) et fIn (Flux de chaleur Q)). On présente dans la figure 3.4 une comparaison entre la simulation monolithique du modèle et sa co-simulation avec l'orchestrateur à pas fixe 'ma' d'OMS.

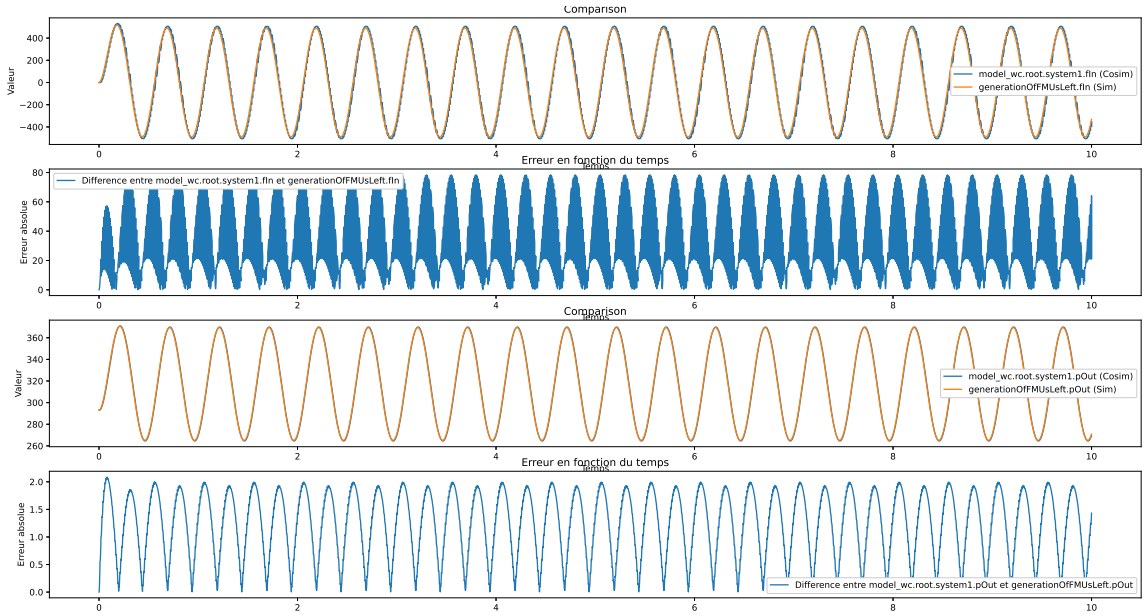


FIG. 3.4 : Comparaison de l'orchestrateur d'OMS avec une simulation monolithique du modèle 3.3

Dans la figure 3.5, nous présentons une comparaison similaire en utilisant notre orchestrateur Aitken-Schwarz. Les deux figures incluent également l'erreur absolue entre l'algorithme souhaité et la simulation monolithique du modèle. Les paramètres préfixés par "generationOffFMUs" proviennent de la simulation monolithique, tandis que ceux commençant par "model_wc.root.systemX" sont issus de la co-simulation.

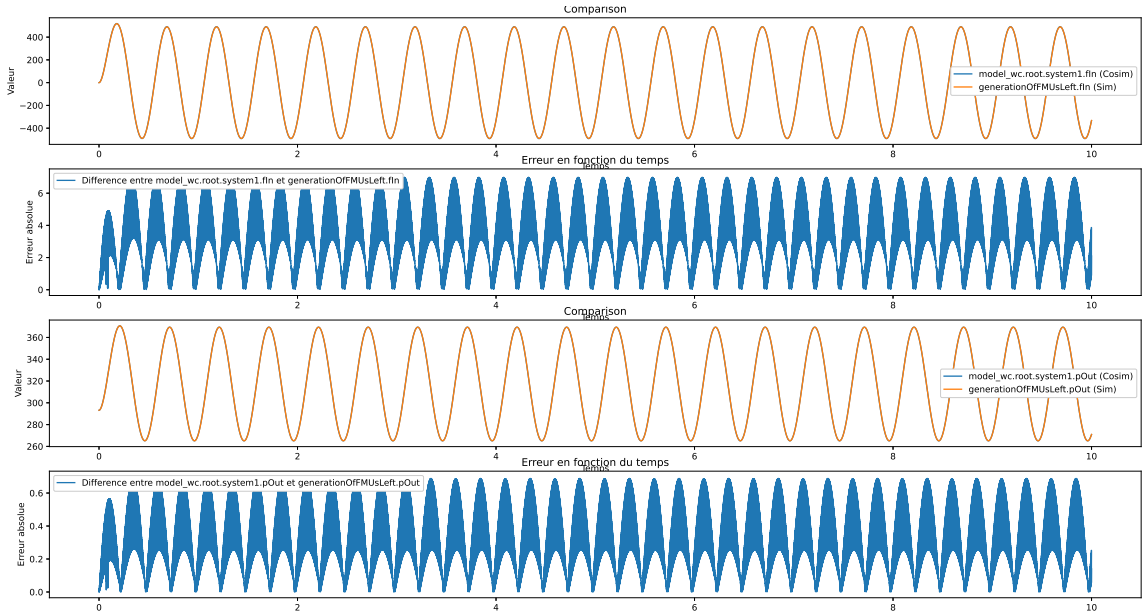


Fig. 3.5 : Comparaison de l'orchestrateur Aitken-Schwarz avec une simulation monolithique du modèle 3.3

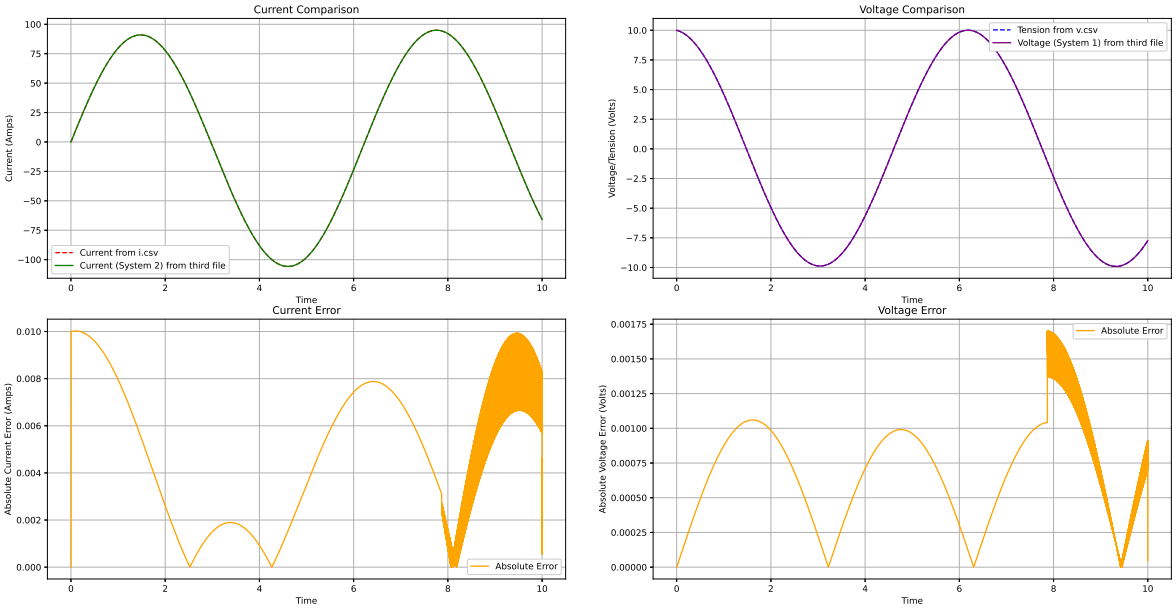


Fig. 3.6 : Comparaison de l'orchestrateur Aitken-Schwarz avec une simulation monolithique du circuit RL

On a aussi testé la méthode sur un circuit RL :

$$V_0 \cos(t) = RI + L\frac{dI}{dt} \tag{3.6}$$

Où le circuit est divisé en deux FMUs : l'un comprend la source de tension et la résistance, et l'autre contient l'inductance. Les résultats de cette configuration sont exposés dans la figure 3.6. Les variables portant le suffixe (System) proviennent de la co-simulation. L'erreur absolue entre les résultats de la co-simulation et ceux de la simulation monolithique est également présentée.

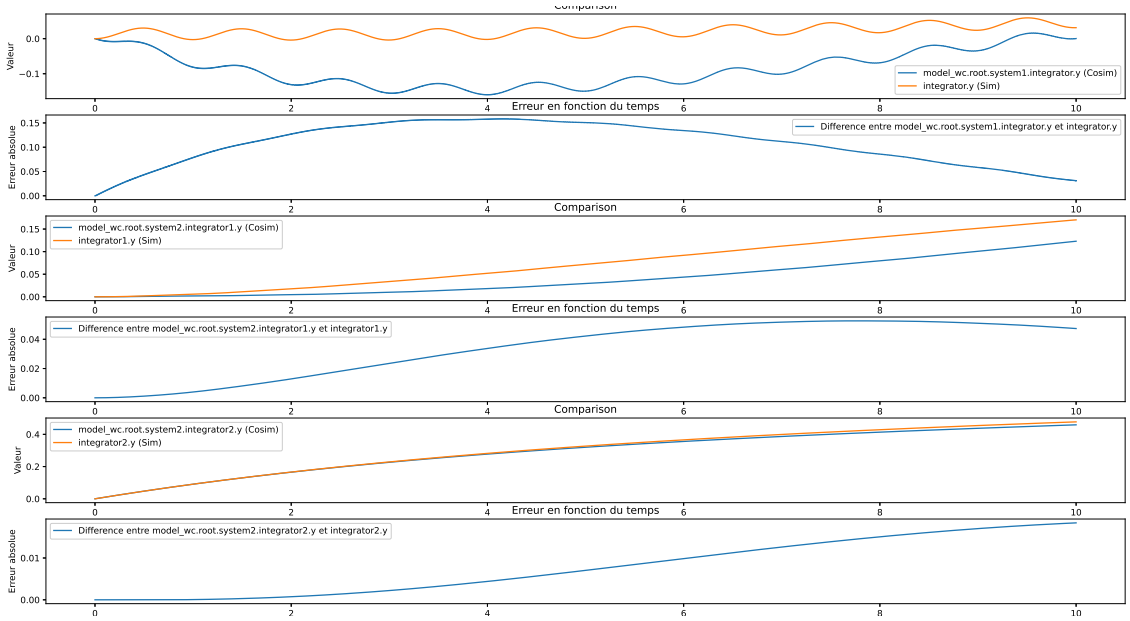


FIG. 3.7 : Comparaison de l'orchestrateur Aitken-Schwarz avec une simulation monolithique du modèle 3.9

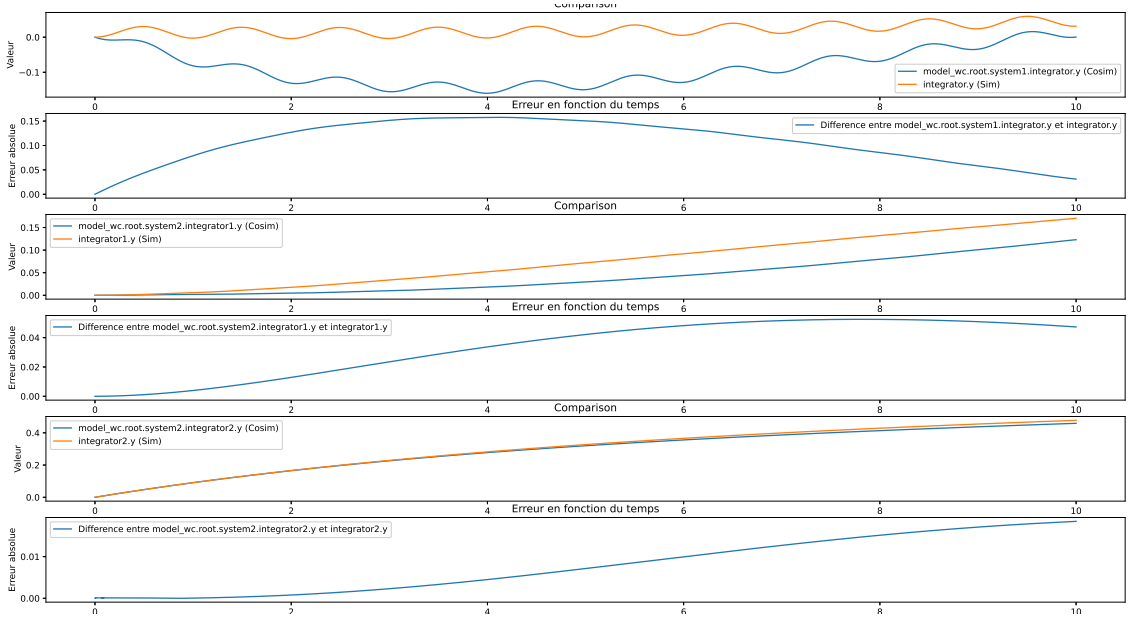


FIG. 3.8 : Comparaison de l'orchestrateur d'OMS avec une simulation monolithique du modèle 3.9

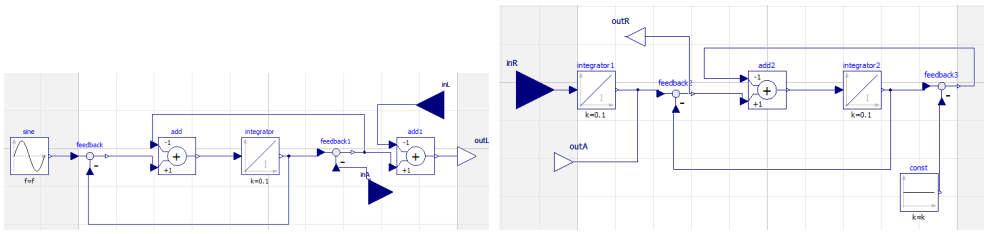


FIG. 3.9 : Modèle utilisés dans la co-simulation d'une chaîne de contrôle de signal

La figure 3.9 représente une chaîne de traitement d'un signal, que l'on a découpée et exportée en FMUs, afin de comparer la simulation monolithique à la co-simulation avec la méthode d'Aitken-Schwarz dans 3.7 et l'orchestrateur à pas fixe d'OMS dans 3.8.

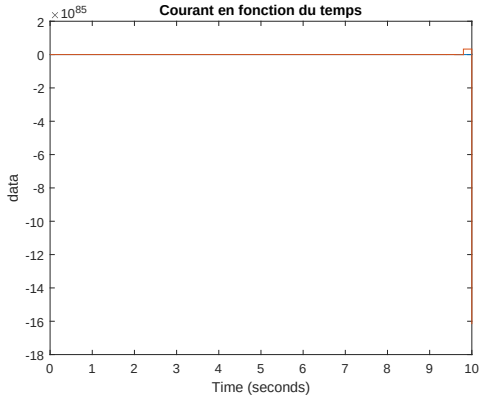


FIG. 3.10 : Résultats de la co-simulation du circuit RL sur Simulink

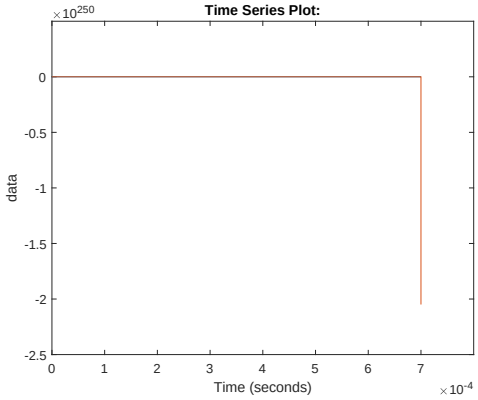


FIG. 3.11 : Résultats de la co-simulation de l'amplificateur différentiel 3.1

La figure 3.4.3 illustre les résultats obtenus via la co-simulation utilisant l'orchestrateur de SIMULINK. Cet orchestrateur appartient à la catégorie des algorithmes qui ne recourent pas à des fonctionnalités avancées telles que la capacité de faire un rollback, la modification des dérivées ou la sérialisation des données.

3.5 Conclusion et perspectives

En conclusion de ce chapitre, nous avons exploré les plateformes de co-simulation et les algorithmes d'orchestration, mettant en lumière leur importance cruciale dans la gestion efficace et précise des simulations complexes. À travers une analyse comparative des plateformes open source disponibles, nous avons identifié OMSimulator comme étant particulièrement adapté à nos besoins en raison de sa robustesse, sa documentation exhaustive, et sa capacité à intégrer diverses configurations de co-simulation. L'implémentation de l'algorithme d'orchestration Aitken-Schwarz a démontré des avantages significatifs en termes de gestion des erreurs et de la stabilité des simulations, confirmant ainsi l'efficacité de notre approche sélectionnée.Effectivement, on observe que l'erreur présentée dans la figure 3.4 atteint une magnitude de 0.14, alors que dans la figure 3.5, elle est réduite à 0.012. Cette réduction montre que la méthode Aitken-Schwarz améliore les performances de la simulation par un facteur de 10 par rapport à la méthode par défaut d'OMS. Par ailleurs, dans la figure 3.6, l'erreur est de $1.05 \cdot 10^{-4}$. Cependant, la figure 3.4.3 révèle que la co-simulation a divergé, en raison de la présence d'une boucle algébrique (DAE), illustrant ainsi les avantages significatifs de la méthode Aitken-Schwarz. Pourtant, comme remarqué dans les figures 3.7, 3.8, les deux orchestrateurs ne parviennent pas à égaler les performances de la simulation monolithique. Cette limitation est attribuable au caractère très dynamique du système et à la manière dont il a été segmenté. En effet, le système comprend plusieurs intégrateurs, soulignant ainsi la nécessité d'adopter une méthode prédictive pour améliorer la précision de la co-simulation.

Conclusion générale et intérêts

Pour conclure ce mémoire, notre exploration approfondie des plateformes de co-simulation et des algorithmes d'orchestration a non seulement démontré leur potentiel transformateur mais aussi leur valeur ajoutée substantielle pour l'équipe RT-sim. L'adoption d'OMSimulator, offre à l'équipe une plateforme de co-simulation plus robuste, facilitant ainsi l'intégration et les tests de nouvelles approches. Cette amélioration marque un tournant, permettant à l'équipe RT-sim de gagner un temps précieux lors de l'incorporation et des essais des innovations.

L'introduction d'une taxonomie basée sur des principes ontologiques s'avère particulièrement bénéfique pour les nouveaux arrivants au projet, car elle facilite la segmentation et la compréhension du sujet.Cette structure bien définie facilite la compréhension des interactions et des rôles des différentes entités impliquées, simplifiant ainsi la recherche bibliographique pour les nouveaux membres du projet.

En outre, nous avons mis en lumière la valeur substantielle des plateformes de co-simulation et des algorithmes d'orchestration, soulignant particulièrement leur rôle crucial dans la gestion efficace et précise de simulations complexes. À travers une analyse comparative des plateformes open source disponibles, OMSimulator s'est distingué comme étant particulièrement adapté à nos besoins grâce à sa robustesse, et sa documentation exhaustive. La mise en œuvre de l'algorithme d'orchestration Aitken-Schwarz a prouvé son efficacité, améliorant significativement la gestion des erreurs et la stabilité des simulations, validant ainsi l'efficacité de notre approche choisie. Cependant, il a été observé que dans certains scénarios, la méthode Aitken-Schwarz n'a pas réussi à produire de bons résultats. Cela met en évidence l'intérêt pour des méthodes prédictives telles que la méthode COSTARICA [36], qui utilise des techniques avancées pour estimer et corriger les erreurs en temps réel.

Les progrès réalisés dans le domaine de la co-simulation peuvent positionner cette technologie comme une alternative supérieure à la simulation monolithique, notamment pour les applications qui nécessitent une collaboration interdisciplinaire. Cela souligne l'importance et l'impact des recherches menées par l'équipe RT-SIM de Capgemini, qui œuvre à optimiser et à étendre les capacités de la co-simulation. En intégrant des avancées telles que les méthodes Aitken-Schwarz, ces recherches contribuent à transformer la co-simulation en un outil plus flexible, précis et efficace, adapté aux défis complexes des systèmes modernes. Cette orientation vers des solutions innovantes illustre l'engagement de l'équipe à fournir des solutions de pointe qui améliorent non seulement la performance mais aussi l'intégrabilité des différents systèmes dans des contextes multidisciplinaires.

Bibliographie

[1] C. Gomes, C. Thule, P.G. Larsen, J. Denil, H. Vangheluwe, Co-simulation of continuous systems : a tutorial, 2018, arXiv preprint arXiv :1809.08463

[2] C. Thule, C. Gomes, J. Deantoni, P.G. Larsen, J. Brauer, H. Vangheluwe, Towards the verification of hybrid co-simulation algorithms, in : Federation of International Conferences on Software Technologies : Applications and Foundations, Springer, 2018, pp. 5–20.

[3] Andreas Abel, Torsten Blochwitz, Alexander Eichberger, Peter Hamann, and Udo Rein. Functional mock-up interface in mechatronic gearshift simulation for commercial vehicles. In 9th International Modelica Conference. Munich, 2012.

[4] D. Bian, M. Kuzlu, M. Pipattanasomporn, S. Rahman, and Y. Wu. Real-time co-simulation platform using OPAL-RT and OPNET for analyzing smart grid performance. In 2015 IEEE Power & Energy Society General Meeting, pages 1–5. IEEE, jul 2015. ISBN 978-1-4673-8040-9.

[5] John Fitzgerald, Carl Gamble, Richard Payne, Peter Gorm Larsen, Stylianos Basagiannis, and Alie El-Din Mady. Collaborative Model-based Systems Engineering for Cyber-Physical Systems – a Case Study in Building Automation. In INCOSE 2016, Edinburgh, Scotland, jul 2016.

[6] Nicolai Pedersen, Jan Madsen, and Morten Vejlgaard-Laursen. Co-Simulation of Distributed Engine Control System and Network Model using FMI & SCNSL. IFAC-PapersOnLine, 48 (16) :261–266, 2015. ISSN 24058963.

[7] Michal Kudelski, Luca M. Gambardella, and Gianni A. Di Caro. RoboNetSim : An integrated framework for multi-robot and network simulation. Robotics and Autonomous Systems, 61 (5) :483–496, may 2013. ISSN 09218890.

[8] The leading standard to exchange dynamic simulation models. Functional Mock-up Interface. (n.d.). https://fmi-standard.org/

[9] Cláudio Gomes, Casper Thule, David Broman, Peter Gorm Larsen and Hans Vangheluwe. Co-simulation : State of the art, 2017; arXiv :1702.00686.

[10] Martin Arnold and Michael Gunther. Preconditioned Dynamic Iteration for Coupled Differential-Algebraic Systems. BIT Numerical Mathematics, 41(1) :1–25, 2001. doi : 10.1023/ A :1021909032551.

[11] B. Schweizer and D. Lu. Predictor/corrector co-simulation approaches for solver coupling with algebraic constraints. ZAMM - Journal of Applied Mathematics and Mechanics, 95(9) :911–938, sep 2015. ISSN 00442267. doi : 10.1002/zamm.201300191.

[12] Bei Gu and H H Asada. Co-simulation of algebraically coupled dynamic subsystems. In American Control Conference, 2001. Proceedings of the 2001, volume 3, pages 2273–2278 vol.3, 2001. ISBN 0743-1619 VO - 3. doi : 10.1109/ACC.2001.946089.

[13] Adelinde M. Uhrmacher. Variable structure models : autonomy and control answers from two different modeling approaches. In 4th Annual Conference on AI, Simulation and Planning in High Autonomy Systems, pages 133–139. IEEE Comput. Soc. Press, 1993. ISBN 0-8186-40200. doi : 10.1109/AIHAS.1993.410588.

[14] R. Kübler and W. Schiehlen. Two Methods of Simulator Coupling. Mathematical and Computer Modelling of Dynamical Systems, 6(2) :93–113, jun 2000. ISSN 1387-3954. doi : 10.1076/1387-3954(200006)6:2;1-M ;FT093.

[15] Simon Thrane Hansen, Casper Thule, Cláudio Gomes, Kenneth Guldbrandt Lausdahl, Frederik Palludan Madsen, Giuseppe Abbiati, Peter Gorm Larsen, Co-simulation at different levels of expertise with Maestro2, Journal of Systems and Software,Volume 209,2024,111905,ISSN 0164-1212, https ://doi.org/10.1016/j.jss.2023.111905.

[16] Ochel, L., Braun, R., Thiele, B., Asghar, A., Buffoni, L., Eek, M., Fritzson, P., Fritzson, D., Horkeby, S., Hällquist, R., Kinnander, Å., Palanisamy, A., Pop, A., & Sjölund, M. (2019). Omsimulator - integrated FMI and TLM-based co-simulation with composite model editing and SSP. Linköping Electronic Conference Proceedings. https ://-doi.org/10.3384/ecp1915769

[17] Évora Gómez, J., Hernández Cabrera, J. J., Tavella, J.-P., Vialle, S., Kremers, E., & Frayssinet, L. (2019). Daccosim ng : Co-simulation made simpler and faster. Linköping Electronic Conference Proceedings. https ://doi.org/10.3384/ecp19157785

[18] Andersson, C., Åkesson, J., & Führer, C. (2016). PyFMI : A Python Package for Simulation of Coupled Dynamic Models with the Functional Mock-up Interface. (Technical Report in Mathematical Sciences ; Vol. 2016, No. 2). Centre for Mathematical Sciences, Lund University.

[19] Dempsey, M. (2006). Dymola for multi-engineering modelling and Simulation. 2006 IEEE Vehicle Power and Propulsion Conference. https://doi.org/10.1109/vppc.2006.364294

[20] Vanfretti, L., Baudette, M., Amazouz, A., Bogodorova, T., Rabuzin, T., Lavenius, J., & Goméz-López, F. J. (2016). Rapid : A modular and extensible toolbox for parameter estimation of Modelica and FMI compliant models. SoftwareX, 5, 144–149. https ://doi.org/10.1016/j.softx.2016.07.004

[21] Perabo, F., Park, D., Zadeh, M. K., Smogeli, O., & Jamt, L. (2020). Digital twin modelling of ship power and propulsion systems : Application of the open simulation platform (OSP). 2020 IEEE 29th International Symposium on Industrial Electronics (ISIE).https://doi.org/10.1109/isie45063.2020.9152218

[22] Arnold, M., Clauß, C., Schierz, T.: Error analysis and error estimates for co-simulation in FMI for model exchange and co-simulation v2.0. In : Sch¨ops, S., Bartel, A., G¨unther, M., ter Maten, E.J.W., M¨uller, P.C. (eds.) Progress in DifferentialAlgebraic Equations. DEF, pp. 107–125. Springer, Heidelberg (2014). https ://doi.org/10.1007/978-3-662-44926-4_6

[23] Hansen, S. T., Thule, C., & Gomes, C. (2021). An FMI-Based Initialization Plugin for INTO-CPS Maestro 2. In L. Cleophas, & M. Massink (Eds.), Software Engineering and Formal Methods : SEFM 2020 Collocated Workshops (pp. 295-310). Springer. https ://doi.org/10.1007/978-3-030-67220-1_22

[24] Alfredo Garro and Alberto Falcone. On the integration of HLA and FMI for supporting interoperability and reusability in distributed simulation. In Spring Simulation Multi-Conference, pages 774–781. Society for Computer Simulation International, 2015.

[25] Tavella JP., Caujolle M., Tan M.,Plessis G., Schumann M., Vialle S., Dad C., et al. Toward an Hybrid Co-simulation with the FMI-CS Standard, apr 2016. https://hal-centralesupelec.archives-ouvertes.fr/hal-01301183.

[26] Jean-Sebastien Bolduc and Hans Vangheluwe. Mapping odes to devs : Adaptive quantization. In Summer Computer Simulation Conference, pages 401–407. Society for Computer Simulation International ; 1998, 2003. ISBN 0094-7474.

[27] Q. Alfalouji et al., « Co-simulation for buildings and smart energy systems — A taxonomic review », Simul. Model. Pract. Theory, vol. 126, p. 102770, juill. 2023, doi : 10.1016/j.simpat.2023.102770.

[28] N. Guarino et C. Welty, « Ontological Analysis of Taxonomic Relationships », in Conceptual Modeling — ER 2000, A. H. F. Laender, S. W. Liddle, et V. C. Storey, Éd., Berlin, Heidelberg : Springer, 2000, p. 210-224. doi : 10.1007/3-540-45393-8_16.

[29] D. Krob, « Éléments de systémique. Architecture des systèmes », in Complexité-Simplexité, A. Berthoz et J.-L. Petit, Éd., Collège de France, 2014. doi : 10.4000/books.cdf.3388.

[30] Hatledal, L. I., Zhang, H., & Collonval, F. (2020). Enabling python driven co-simulation models with Pythonfmu. ECMS 2020 Proceedings Edited by Mike Steglich, Christian Mueller, Gaby Neumann, Mathias Walther. https://doi.org/10.7148/2020-0235

[31] Legaard, C., Tola, D., Schranz, T., Macedo, H., & Larsen, P. (2021). A universal mechanism for implementing functional mock-up units. Proceedings of the 11th International Conference on Simulation and Modeling Methodologies, Technologies and Applications. https://doi.org/10.5220/0010577601210129

[32] Héléna Shourick, Damien Tromeur-Dervout and Laurent Chédot. Co-simulation domain decomposition algorithm for hybrid EMT-Dynamic Phasor modeling, 2022; arXiv :2212.05232.

[33] Garbey, M., & Tromeur-Dervout, D. (2002). On some aitken-like acceleration of the Schwarz method. International Journal for Numerical Methods in Fluids, 40(12), 1493–1513. https://doi.org/10.1002/fld.407

[34] Eguillon, Y., Lacabanne, B., Tromeur-Dervout, D.: IFOSMONDI : A Generic Co-simulation Approach Combining Iterative Methods for Coupling Constraints and Polynomial Interpolation for Interfaces Smoothness. In : 9th International Conference on Simulation and Modeling Methodologies, Technologies and Applications. pp. 176–186. SCITEPRESS - Science and Technology Publications, Prague, Czech Republic (Jul 2019),https://doi.org/10.5220/0007977701760186

[35] Busch, M.: Performance Improvement of Explicit Co-simulation Methods Through Continuous Extrapolation. In : IUTAM Symposium on solver-coupling and co-simulation. IUTAM Bookseries, vol. 35, pp. 57–80. IUTAM (2019). https://doi.org/10.1007/978-3-030-14883-6

[36] Eguillon Y.,Lacabanne B. and Tromeur-Dervout D.. COSTARICA estimator for rollback-less systems handling in iterative co-simulation algorithms, 2022; arXiv :2203.11752.

Annexes

Annexe A

Documentation OMSimulator

A.1 Vue d'ensemble d'OMSimulator

1. **AlgLoop.h**
   Rôle : Ces classes gèrent les boucles algébriques dans la simulation, en s'assurant que les dépendances sont correctement résolues et que les équations du système sont résolues efficacement.

2. **BusConnector.h**
   Rôle : Cette classe gère les connexions de bus entre les composants du modèle. Elle gère les interconnexions permettant à différentes parties du modèle de simulation de communiquer entre elles par le biais d'un bus.

3. **Clocks.h**
   Rôle : La classe `Clock` gère les horloges de simulation, essentielles pour synchroniser les événements et les pas de temps pendant le processus de simulation.

4. **ComponentFMUCS.h**
   Rôle : Représente les composants FMUs pour la Co-Simulation. Elle gère l'intégration des FMUs qui suivent la norme de Co-Simulation, leur permettant d'interagir dans l'environnement OMSimulator.

5. **ComponentFMUME.h**
   Rôle : Semblable à `ComponentFMUCS`, cette classe représente les composants FMUs mais spécifiquement pour l'échange de modèles. Elle facilite l'intégration et l'interaction des FMUs conçus pour l'échange de modèles dans la simulation.

6. **ComponentTable.h**
   Rôle : Gère les composants basés sur des tables dans la simulation. Ces composants utilisent des tables prédéfinies pour leur comportement, offrant un moyen simple d'inclure des tables de correspondance dans la simulation.

7. **Connection.h**
   Rôle : La classe `Connection` gère les connexions entre connecteurs. Elle assure que les données circulent correctement entre les différents composants du modèle, maintenant l'intégrité du réseau de simulation.

8. **Connector.h**
   Rôle : Représente les connecteurs des composants, qui sont les points par lesquels les composants interagissent entre eux. Cette classe définit les propriétés et le comportement de ces points d'interaction.

9. **DirectedGraph.h**
   Rôle : Ces classes sont utilisées pour représenter et manipuler les graphes orientés dans les modèles de systèmes. Les graphes dirigés sont essentiels pour modéliser les dépendances et les flux de données.

10. **ExternalModelInfo.h**
    Rôle : Stocke des informations sur les modèles externes intégrés à l'OMSimulator. Cela comprend les métadonnées et les détails de configuration nécessaires pour utiliser correctement les modèles externes.

11. **FMUInfo.h**
    Rôle : Gère les informations liées aux FMUs, y compris leur configuration, leurs paramètres et autres métadonnées. Cette classe est cruciale pour la gestion des FMUs dans la simulation.

12. **Logging.h**
    Rôle : Définit les fonctionnalités de journalisation pour OMSimulator. La journalisation est essentielle pour suivre l'exécution des simulations, le débogage et l'analyse des performances.

13. **Model.h**
    Rôle : Représente le modèle global dans l'OMSimulator. Cette classe encapsule l'ensemble du modèle de simulation, gérant ses composants, connexions et comportement général.

14. **OMSFileSystem.h**
    Rôle : Définit les utilitaires de système de fichiers pour OMSimulator, fournissant des fonctions pour gérer les opérations de fichiers, les répertoires et les chemins, nécessaires pour lire et écrire des données de modèle.

15. **OMSString.h**
    Rôle : Définit les utilitaires de chaîne pour OMSimulator, offrant des fonctions pour gérer les opérations de chaîne, les conversions et les manipulations au sein du logiciel de simulation.

16. **ResultReader.h**
    Rôle : Lit les résultats de la simulation à partir de fichiers de sortie. Cette classe est responsable de l'analyse et de l'interprétation des données générées par les simulations, les rendant accessibles pour l'analyse et la visualisation.

17. **System.h**
    Rôle : Représente les composants systèmes dans le modèle. Cette classe gère la structure hiérarchique de la simulation, permettant des modèles complexes avec plusieurs systèmes imbriqués.

18. **SystemSC.h**
    Rôle : Représente les composants système fortement connectés. Ce sont des systèmes où les composants ont des interdépendances fortes, nécessitant une gestion spécialisée dans la simulation.

19. **SystemWC.h**
    Rôle : Représente les composants système faiblement connectés. Contrairement aux systèmes fortement connectés, ceux-ci ont des interdépendances plus lâches, permettant différentes stratégies d'optimisation et de simulation.

20. **Variable.h**
    Rôle : Gère les variables au sein du modèle, y compris leurs propriétés, valeurs et interactions. Les variables sont fondamentales pour la simulation, représentant l'état et les paramètres des composants du modèle.

A.2 Vue d'ensemble de la classe

La classe `SystemWC` dans OMSimulator traite les systèmes faiblement couplés (FMU-CS), héritant de la classe `System`. Elle offre des fonctionnalités pour gérer les solveurs, initialiser le système, gérer les étapes de simulation et interagir avec les fichiers de résultats. Un schéma de connections entre cette classe et les autres classes d'OMS est présenté dans la figure A.1, donnant ainsi une vue globale sur le Fonctionnement de la plateforme.

A.3 Composants et Méthodes Clés

1. **Constructeur et Destructeur**

   - **Constructeur** : Initialise un nouvel objet `SystemWC`, en définissant la méthode de solveur par défaut et en établissant des références au modèle parent et au système.
   - **Destructeur** : Assure un nettoyage adéquat lorsque un objet `SystemWC` est détruit.

2. **Méthode de Fabrique**

   - **NewSystem** : Crée une nouvelle instance de `SystemWC`. Elle valide l'identifiant et s'assure que le modèle parent ou le système soit fourni, mais pas les deux.

3. **Méthodes de Solveur**

   - **getSolverName** : Renvoie le nom de la méthode de solveur actuelle sous forme de chaîne.
   - **setSolverMethod** : Définit la méthode de solveur en fonction de la chaîne fournie, prenant en charge plusieurs types de solveurs (par exemple, `oms-ma`, `oms-mav`).

4. **Importation/Exportation d'Informations de Simulation**
    - **exportToSSD_SimulationInformation** : Exporte les informations de simulation vers un nœud XML SSD (Description de la Structure du Système), incluant des détails sur la méthode de solveur et ses paramètres.
    - **importFromSSD_SimulationInformation** : Importe les informations de simulation à partir d'un nœud XML SSD, définissant la méthode de solveur et ses paramètres en conséquence.
5. **Méthodes de Cycle de Vie du Système**
    - **instantiate** : Prépare le système pour la simulation en initialisant tous les sous-systèmes et composants.
    - **initialize** : Initialise le système, met à jour les graphes de dépendances et prépare les structures de données pour la simulation.
    - **terminate** : Met fin à tous les sous-systèmes et composants, garantissant que les ressources sont correctement nettoyées.
    - **reset** : Réinitialise le système et ses composants à leur état initial, prêts pour un nouveau cycle de simulation.
6. **Méthodes d'Étape de Simulation**
    - **doStep** : Fait avancer la simulation d'une étape, en implémentant différentes algorithmes basés sur la méthode de solveur. Elle gère les tailles de pas adaptatives et la vérification des erreurs.
    - **stepUntil** : Fait avancer la simulation jusqu'à un temps d'arrêt spécifié, en appelant répétitivement `doStep` selon les besoins.
    - **stepUntilASSC** : Traitement spécial pour la méthode `oms_solver_wc_assc`, incluant la détection et la gestion des événements.
7. **Méthodes Auxiliaires**
    - **getMaxOutputDerivativeOrder** : Renvoie l'ordre le plus élevé de dérivées de sortie parmi les composants du système.
    - **getRealOutputDerivative** : Récupère la dérivée de sortie réelle pour un signal spécifié.
    - **setRealInputDerivative** : Définit la dérivée d'entrée réelle pour un signal spécifié.
    - **registerSignalsForResultFile** : Enregistre les signaux du système avec le fichier de résultats pour enregistrer les sorties de simulation.
    - **updateSignals** : Met à jour le fichier de résultats avec les valeurs actuelles des signaux du système.
8. **Gestion des Entrées/Sorties**
    - **getInputs** : Récupère les valeurs d'entrée pour le système en se basant sur le graphe dirigé des dépendances.

- **setInputsDer** : Définit les dérivées d'entrée pour le système.
- **updateInputs** : Met à jour les entrées du système en fonction du graphe dirigé, garantissant une propagation correcte des valeurs à travers le système.
- **getinputs2** : Récupère un vecteur d'entrées en tenant compte des boucles algébriques au sein du système.
- **updateinputs2** : Met à jour les entrées du système tout en tenant compte des boucles algébriques, garantissant la cohérence.

9. **Méthodes Utilitaires**

- **vectorToString** : Convertit un vecteur en sa représentation en chaîne pour la journalisation ou le débogage.
- **matrixToString** : Convertit une matrice en sa représentation en chaîne pour la journalisation ou le débogage.

A.4 Gestion des boucles algébriques

Les classes `oms::System` et `oms::AlgLoop` gèrent les boucles algébriques en utilisant différentes méthodes de résolution dans un cadre de simulation. Le code inclut des méthodes pour résoudre les boucles algébriques en utilisant l'itération de point fixe et le solveur KINSOL, ainsi que des fonctionnalités pour récupérer et définir des données de simulation, et logger les statuts et erreurs du système. Voici un résumé des fonctionnalités clés :

Méthodes de classe :

- **solveAlgLoop(DirectedGraph& graph, int loopNumber, std ::vector<double>* additionalData)** : Cette méthode redirige la résolution d'une boucle algébrique vers l'objet `AlgLoop` respectif.
- **solveAlgLoop(System& syst, DirectedGraph& graph, std ::vector<double>* additionalData)** : Implémentée dans la classe `oms::AlgLoop`, cette méthode décide de la méthode de solveur en fonction de `algSolverMethod` et gère la boucle en conséquence.
- **fixPointIteration(System& syst, DirectedGraph& graph, additionalData)** : Cette méthode tente de résoudre la boucle en utilisant une approche d'itération de point fixe, mettant à jour et vérifiant les résidus pour la convergence.

Stratégies de solveur :

- **Itération de point fixe** : Vise à résoudre en ajustant itérativement les valeurs jusqu'à ce que les sorties se stabilisent dans des résidus acceptables.

- **KINSOL** : Utilise le solveur KINSOL de la suite SUNDIALS, généralement utilisé pour des systèmes algébriques plus complexes nécessitant des solveurs non linéaires robustes.

Gestion des erreurs et logging :

- Utilisation extensive du logging pour déboguer et suivre la progression de la résolution des boucles algébriques.
- La gestion des erreurs est mise en œuvre pour gérer les situations où la méthode de solveur est invalide ou où le nombre maximal d'itérations est dépassé.

Interaction système :

- Le code gère les états du système en récupérant et en définissant les valeurs réelles basées sur les connexions des nœuds du graphe dirigé, facilitant l'interaction dynamique au sein des composants de simulation.

Efficacité de l'algorithme et contrôle :

- L'implémentation vérifie le nombre d'itérations par rapport à un maximum prédéfini pour éviter les boucles infinies et assurer l'efficacité computationnelle.

Cette structure de code est robuste pour les environnements de simulation modulaires où différentes boucles algébriques peuvent être abordées avec des solveurs appropriés, et elle garantit que chaque partie du système peut être individuellement adressée et gérée pendant le processus de simulation.

A.5 Analyse Détaillée des Méthodes Principales

Constructeur et Destructeur

Le constructeur initialise l'instance de `SystemWC` avec des références à son modèle parent et au système, en définissant la méthode de solveur par défaut. Le destructeur assure que toutes les ressources allouées par le `SystemWC` sont correctement nettoyées lorsque l'objet est détruit.

Méthodes de Solveur

La méthode `getSolverName` renvoie le nom de la méthode de solveur actuelle basée sur la valeur enum du solveur. La méthode `setSolverMethod` permet de définir la méthode de solveur en faisant correspondre une chaîne fournie aux méthodes de solveur prises en charge et en mettant à jour l'état interne du solveur en conséquence.

Importation/Exportation d'Informations de Simulation

La méthode `exportToSSD_SimulationInformation` sérialise la configuration actuelle du solveur et les paramètres dans un nœud XML SSD pour la persistance ou l'interopérabilité. La méthode `importFromSSD_SimulationInformation` désérialise la configuration du solveur à partir d'un nœud XML SSD, garantissant que le système est configuré correctement en fonction des données importées.

Méthodes de Cycle de Vie du Système

La méthode `instantiate` prépare tous les sous-systèmes et composants pour la simulation en les initialisant. La méthode `initialize` met à jour les graphes de dépendances, initialise les entrées et met en place les structures de données nécessaires pour la simulation. La méthode `terminate` assure l'arrêt soigné de tous les sous-systèmes et composants, veillant à ce qu'ils soient correctement fermés. Quant à la méthode `reset`, elle réinitialise l'ensemble du système à son état initial, le préparant ainsi pour un nouveau cycle de simulation.

Méthodes d'Étape de Simulation

La méthode `doStep` fait avancer la simulation d'une étape, utilisant différents algorithmes basés sur la méthode de solveur actuelle. Elle comprend la gestion des tailles de pas adaptatives et la vérification des erreurs. La méthode `stepUntil` fait avancer la simulation jusqu'à un temps d'arrêt spécifié, en appelant répétitivement `doStep` et en mettant à jour l'état de la simulation.

Gestion des Entrées/Sorties

La méthode `getInputs` récupère les valeurs d'entrée du système en se basant sur le graphe dirigé des dépendances, les préparant pour une utilisation dans la simulation. La méthode `setInputsDer` définit les dérivées d'entrée pour le système, permettant une gestion dynamique des entrées. La méthode `updateInputs` met à jour les entrées du système en fonction du graphe dirigé, garantissant que les valeurs sont propagées correctement. La méthode `getinputs2` récupère des entrées tout en tenant compte des boucles algébriques, garantissant la cohérence dans les systèmes avec des boucles de rétroaction. La méthode `updateinputs2` met à jour les entrées tout en tenant compte des boucles algébriques, garantissant que l'état du système reste cohérent.

TAB. A.1 : Script d'un scénario de co-simulation

```
from OMSimulator import OMSimulator

oms = OMSimulator()
oms.setTempDirectory("./temp/")
oms.setLoggingLevel(2)
oms.newModel("model_wc")

oms.addSystem("model_wc.root", oms.system_wc)
oms.setResultFile('model_wc', 'DifferenceAmplifier.csv')

# Add sub models for System1.fmu and System2.fmu
oms.addSubModel("model_wc.root.system1", ...
    "~/DifferenceAmplifierLeft.fmu")
oms.addSubModel("model_wc.root.system2", ...
    "~/DifferenceAmplifierRight.fmu")

# Add connections between System1 and System2
oms.addConnection("model_wc.root.system1.vout", ...
    "model_wc.root.system2.vin")
oms.addConnection("model_wc.root.system1.iin", ...
    "model_wc.root.system2.iout")

 # Set Simulation settings (tolerance 1e-6)
oms.setTolerance('model_wc', 1e-6, 1e-6)
oms.setSolver('model_wc', oms.solver_wc_mav)
oms.setVariableStepSize('model_wc', 1e-3, 1e-6,1e-1)
oms.setStopTime('model_wc', 10)

# Logging setup
oms.setLogFile("~/test.log")

# instantiate and set start values to variables
oms.instantiate('model_wc')

# Initialize, simulate and terminate the model
oms.initialize("model_wc")
oms.simulate("model_wc")
oms.terminate("model_wc")
oms.delete("model_wc")
```

TAB. A.2 : Script de génération d'un FMU à l'aide de pythonfmu

```
from pythonfmu import Fmi2Causality, Fmi2Variability, ...
    Fmi2Slave, Real
import math

class Source(Fmi2Slave):

def __init__(self, **kwargs):
    super().__init__(**kwargs)
    self.Vmax = 10.0
    self.i1 = 0.0
    self.v1 = self.Vmax
    self.Vt = 0.0
    self.R = 10000.
    self.register_variable(Real("Vmax", ...
        causality=Fmi2Causality.parameter, ...
        variability=Fmi2Variability.tunable))
    self.register_variable(Real("i1", ...
        causality=Fmi2Causality.input))
    self.register_variable(Real("R", ...
        causality=Fmi2Causality.parameter, ...
        variability=Fmi2Variability.tunable))
    self.register_variable(Real("Vt", ...
        causality=Fmi2Causality.local, ...
        variability=Fmi2Variability.tunable))
    self.register_variable(Real("v1", ...
        causality=Fmi2Causality.output))

def do_step(self, current_time, step_size):
    self.Vt = self.Vmax * math.cos(current_time+step_size)
    self.v1 = self.Vt - self.i1*self.R
return True
```

```
import numpy as np
from concurrent.futures import ThreadPoolExecutor
import matplotlib.pyplot as plt
import pandas as pd

class CoSimulation:
    def __init__(self, A, B, initial_state, t_final, dt, ...
       max_iter=100, tol=0.000001):
        self.A = A
        self.B = B
        self.state = initial_state
        self.t_final = t_final
        self.dt = dt
        self.max_iter = max_iter
        self.tol = tol
        self.time = 0
        self.history = []
        self.original_dt = dt

    def initialize(self, initial_time, initial_state):
        self.time = initial_time
        self.state = initial_state
        self.history = [(self.time, self.state.copy())]
        print(f"Initialized simulation at time {self.time} with ...
           state {self.state}")

    def do_step(self):
        # Store the previous state and time in case of rollback
        prev_state = self.state.copy()
        prev_time = self.time

        # Update S with the current time
        S = np.array([0.0, 10 * np.cos(self.time +dt)])
        print(f"Time {self.time:.4f}: Computing next state with ...
           X = {self.state}")

        # Use Aitken-Schwarz method with acceleration to ...
           compute the next state
        converged, self.state = ...
           aitken_schwarz_with_acceleration(self.A, self.B, S, ...
           self.state, self.max_iter, self.tol)

        if not converged :
            if self.dt / 10 >= min_dt:
                print(f"Max iterations exceeded or result not ...
                   within tolerance. Rolling back and reducing ...
                   timestep from {self.dt} to {self.dt / 10}.")
                self.state = prev_state
```

```
                self.time = prev_time
                self.dt /= 10
                self.do_step()  # Recursive call with reduced ...
                    timestep
            else:
                raise RuntimeError("Minimum timestep reached. ...
                    Convergence not achieved.")
        else:
            self.time += self.dt
            self.history.append((self.time, self.state.copy()))
            print(f"Time {self.time:.4f}: Updated state to {...
                self.state}")
            if self.dt != self.original_dt:
                print(f"Convergence achieved. Resetting ...
                    timestep to original value {self.original_dt}.")
                self.dt = self.original_dt

    def run_simulation(self):
        while self.time < self.t_final:
            self.do_step()
        print("Simulation complete.")

    def save_results_to_csv(self, filename):
        times = [t for t, _ in self.history]
        states = np.array([state for _, state in self.history])
        data = {'Time': times}
        for i in range(states.shape[1]):
            data[f'State {i}'] = states[:, i]
        df = pd.DataFrame(data)
        df.to_csv(filename, index=False)
        print(f"Results saved to {filename}")

# Function to compute X_i[n+1]^(m+1)
def compute_X_i(i, A, B, S, X_curr, X_next, n):
    Aii_inv = 1 / A[i, i]
    sum_BX = np.sum(B[i, :] * X_curr)
    sum_A_lower = np.sum(A[i, :i] * X_next[:i])
    sum_A_upper = np.sum(A[i, i+1:] * X_next[i+1:])
    return Aii_inv * (sum_BX - sum_A_lower - sum_A_upper + S[i])

# Function for one Schwarz iteration
def schwarz_iterate(A, B, S, X, X_next, n):
    with ThreadPoolExecutor(max_workers=n) as executor:
        futures = [
            executor.submit(compute_X_i, i, A, B, S, X, X_next, n)
            for i in range(n)
        ]
        for i, future in enumerate(futures):
            X_next[i] = future.result()
```

```
    return X_next
# Function to perform the Aitken-Schwarz method with Aitken acceleration
def aitken_schwarz_with_acceleration(A, B, S, X, max_iter=100, tol=0.000001):
    n = len(X)
    errors = []
    iter_num = 0
    P_saved = None
    # Perform the first double iteration to get initial X and X_next
    initial_X = X.copy()
    while iter_num < max_iter:
        print(f"Iteration {iter_num + 1}: Starting with X = {X} ")
        X_next = X.copy()
        X_next1 = schwarz_iterate(A, B, S, X, X_next, n)
        X_next = schwarz_iterate(A, B, S, X, X_next1, n)
        iter_num += 1
        # Compute error
        err = X_next - X
        errors.append(err)
        print(f"Iteration {iter_num}: Error = {np.linalg.norm(err)}")
        if np.linalg.norm(err) > tol :
        # Perform Aitken acceleration if we have enough errors
            if len(errors) > n :
                if P_saved is None:  # Calculate P matrix if not saved
                    err_matrix = np.column_stack(errors[-n:])
                    err_matrix_prev = np.column_stack(errors[-(n+1):-1])

                    try:
                        P = np.dot(err_matrix, np.linalg.inv(err_matrix_prev))
                        P_saved = P  # Save P matrix
                    except np.linalg.LinAlgError:
                    # If the matrix inversion fails, skip Aitken acceleration for this iteration
                        print(f"Iteration {iter_num}: Aitken acceleration skipped due to LinAlgError")
                        P_saved = None
                        continue
            # Generalized Aitken formula
                I = np.eye(n)
                try:
                # Calculate X_next_inf using the original X_next and X from iteration 0
```

```
124                     X_next_inf = np.dot(np.linalg.inv(I - ...
                            P_saved), (X_next - np.dot(P_saved, ...
                            initial_X)))
125                     print(f"Iteration {iter_num}: Aitken ...
                            acceleration applied")
126                 # Check for convergence
127                     final_error = np.linalg.norm(X_next_inf - ...
                            initial_X)
128                     print(f"Iteration {iter_num}: Final error ...
                            after Aitken acceleration = {final_error}")
129                     if final_error < tol:
130                         return True, X_next_inf
131                     else:
132                         print(f"Iteration {iter_num}: Final ...
                                error not within tolerance, ...
                                recalculating P matrix.")
133                         P_saved = None  # Erase saved P matrix
134                 except np.linalg.LinAlgError:
135                 # If the matrix inversion fails, skip this step
136                     print(f"Iteration {iter_num}: Aitken ...
                            acceleration skipped due to LinAlgError")
137                     P_saved = None
138             else:
139                 iter_num += 1
140                 X = X_next
141         else :
142             return True, X_next
143     print(f"Exceeded maximum Schwarz iterations ({max_iter}) ...
            without convergence.")
144     return False, X
145 # Example usage
146 n = 2
147 Zs = 0.001  # Example value for Zs
148 L = 0.0000007   # Example value for L
149 dt = 0.002  # Time step size
150 T = 10  # Total time for simulation
151 min_dt = 0.000002  # Minimum step size
152 A = np.array([[1, 0],[Zs, 1]])
153 B = np.array([[1, -dt/L],[0, 0]])
154 initial_state = np.array([0.0, 0.0])
155 t_final = 10
156 cosim = CoSimulation(A, B, initial_state, t_final, dt)
157 cosim.initialize(0, initial_state)
158 cosim.run_simulation()
159 cosim.save_results_to_csv('cosimulation_results.csv')
```

Listing A.1: méta-modèle d'une plateforme de co-simulation

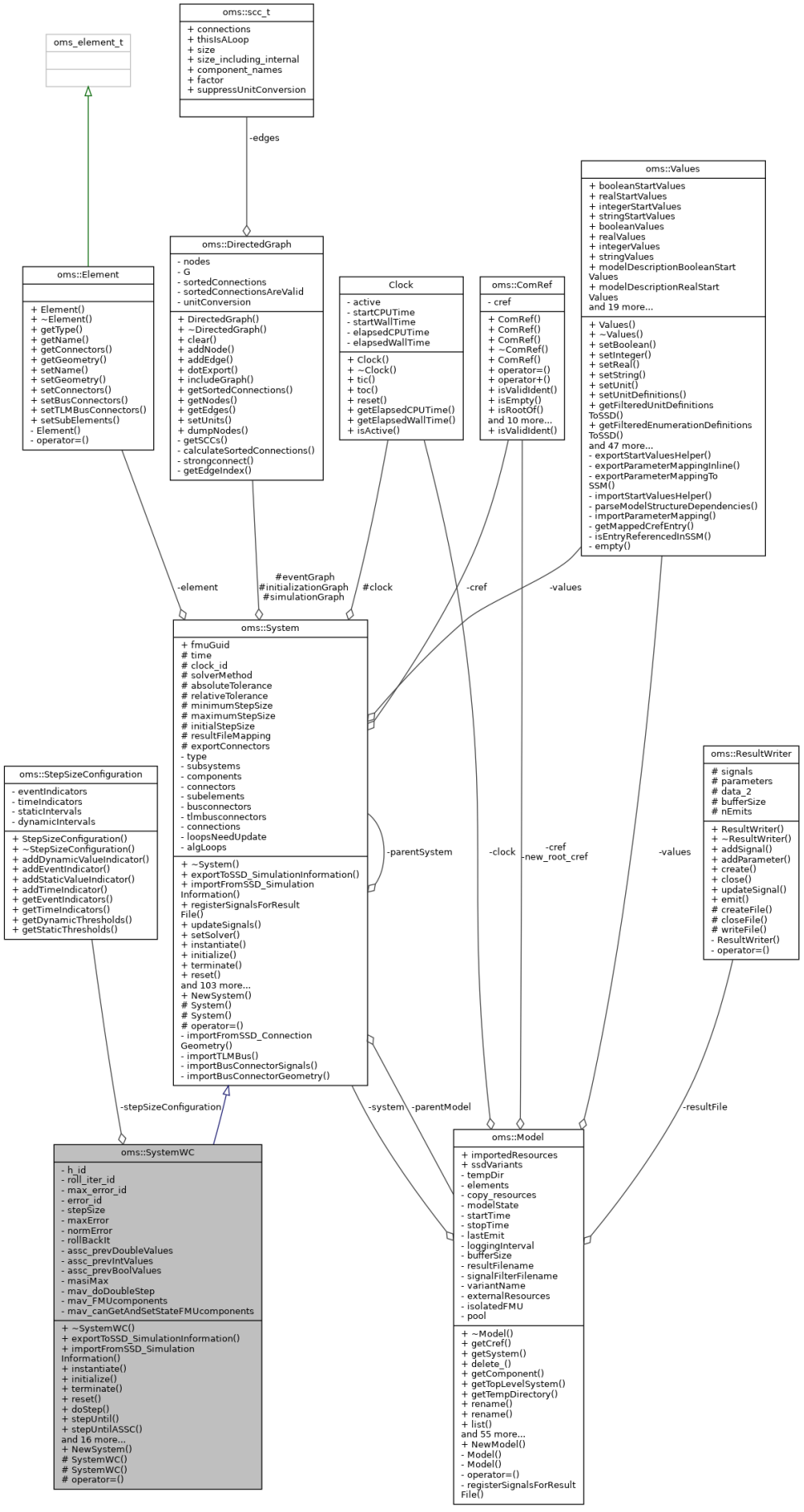


Fig. A.1 : Diagramme de connections de la classe SystemWC

Tab. A.3 : Pseudo-algorithme

```
Result : Do-simulation avec ajustement dynamique de la taille de pas utilisant
         'oms_solver_wc_mav'
Fonction EtapeSimulation(FMUs, time, stepSize, maxTime) :
    tNext ← time + stepSize
    if tNext > maxTime then
        tNext ← maxTime
        stepSize ← tNext − time
    foreach FMU dans FMUs do
        FMU.SauvegarderEtat()
        if non FMU.stepUntil(tNext) then
            return Faux
    return Vrai
Fonction AjusterTailleDePas(inputVect, outputVect, maxError,
 safetyFactor) :
    error ← calculer l'erreur entre inputVect et outputVect
    normError ← calculer la norme de error
    if normError > maxError then
        stepSize ← stepSize(safetyFactorpow(1.0/normError, 0.5))
        if stepSize < minimumStepSize then
            stepSize ← minimumStepSize
        RestaurerEtat(FMUs)
        return Faux
    return Vrai
Fonction SauvegarderEtat(FMUs) :
    foreach FMU dans FMUs do
        FMU.saveState()
Fonction RestaurerEtat(FMUs) :
    foreach FMU dans FMUs do
        FMU.restoreState()
while time < maxTime do
    if EtapeSimulation(FMUs, time, stepSize, maxTime) then
        if AjusterTailleDePas(inputVect, outputVect, maxError, safetyFactor)
         then
            time ← tNext
        else
            Continue la boucle avec la taille de pas ajustée
    else
        return Faux, time
return Vrai, time
```

TAB. A.4 : Pseudo-algorithme

```
Result : Exécuter une simulation avec une méthode à pas fixe utilisant
         'oms_solver_wc_ma'
1  Fonction SimulerEtape(time, maximumStepSize, stopTime) :
2      tNext ← time + maximumStepSize
3      if tNext > stopTime then
4          tNext ← stopTime
5      h ← tNext − time
6      if masiMax > 1 then
7          for component in getComponents() do
8              component.SauvegarderEtat()
9      ObtenirEntrées(eventGraph, inputVect1)
10     for masi from 0 to masiMax - 1 do
11         if useThreadPool() then
12             Parallel execution of subsystems and components using
                thread pool
13         else
14             Serial execution of subsystems and components
15         if masi < masiMax - 1 then
16             MettreAJourEntrées(eventGraph)
17             ObtenirEntrées(eventGraph, inputVect2)
18             inputDer.clear()
19             for inputI from 0 to inputVect1.size() - 1 do
20                 inputDer.push_back((inputVect2[inputI] - inputVect1[inputI]) / h)
21             for component in getComponents() do
22                 component.RestaurerEtat()
23             DéfinirDérivéesEntrées(eventGraph, inputDer)
24         else
25             time ← tNext  MettreAJourEntrées(eventGraph)
26     return oms_status_ok
```

Tab. A.5 : Pseudo-algorithme

```
Result : Run a dynamic step-size simulation using
Initialize simulation parameters
while time < stopTime do
    tNext ← time + stepSize
    if tNext > stopTime then
        tNext ← stopTime
        stepSize ← tNext − time
    foreach component in FMU components do
        component.SaveState()
    iterationNum ← 0
    while iterationNum < 100 do
        GetInputs(eventGraph, inputVector)
        StepUntil(tNext)
        GetInputs(eventGraph, outputVector)
        error ← CalculateError(inputVector, outputVector)
        LogDebug(Error calculation and logging)
        if error < tolerance then
            UpdateInputs(eventGraph, outputVector)
            Emit(time, true)
            LogDebug(Convergence achieved, updating and emitting)
            break
        LogDebug(Preparing for next iteration)
        iterationNum ← iterationNum + 1
    if error ≥ tolerance then
        RestoreState(FMU components)
        LogDebug(Restoring state due to non-convergence)
        Reduce stepSize
    time ← tNext
    SetFmuTime(FMU components, time)
LogDebug(End of simulation)
```