UNIVERSITÉ PARIS SUD

Comprendre le monde, construire l'avenir®

cea list

energie atomique • energies alternatives

UNIVERSITE PARIS-SUD

ÉCOLE DOCTORALE : EDIPS

Laboratoire d'Ingénierie dirigée par les modèles pour les Systèmes Embarqués (LISE), CEA LIST

***DISCIPLINE** informatique*

THÈSE DE DOCTORAT

soutenue le 26/09/2012

par

Mohamed-Lamine BOUKHANOUFA

Adaptabilité et reconfiguration des systèmes temps-réel embarqués

Directeur de thèse :	François TERRIER	Professeur (CEA LIST)
Encadrant de thèse :	Ansgar RADERMACHER	Docteur (CEA LIST)

Composition du jury :

Président du jury :	Alain MERIGOT	Professeur (Université Paris-sud)
Rapporteurs :	Laurent PAUTET	Professeur (TELECOM ParisTech)
	Jean-Michel BRUEL	Professeur (Université de Toulouse)
Examinateurs :	Alain MERIGOT	Professeur (Université Paris-sud)
	Lionel SEINTURIER	Professeur (Université Lille 1)

Résumé

Les systèmes temps réel peuvent être grands, distribués et avoir un environnement dynamique. Cela exige la mise en place de différents modes de fonctionnement et techniques de fiabilité. Par ailleurs, ces différents changements dynamiques d'architecture et de comportement ont un impact sur les caractéristiques temporelles des systèmes qui nécessitent une étude particulière de la capacité des comportements d'adaptation à garantir les contraintes fixées aux systèmes. Ces comportements d'adaptation amplifient la complexité du développement des systèmes temps réel. L'ingénierie dirigée par les modèles, si elle aide à maîtriser le développement de grands systèmes, n'intègre pas bien le volet comportement adaptatif des systèmes temps réel. Le travail présenté dans cette thèse est focalisé sur la spécification de l'adaptabilité d'un système temps réel et l'étude sur de jeux de configurations prédéfinis de l'impact temporel des actions d'adaptation dynamique.

Pour cela, nous présentons une méthodologie outillée basée sur la notion de Mode (extension de machine à état UML) du profil MARTE. Chaque mode représente un comportement possible du système pour un environnement bien déterminé associé à une configuration logicielle. Le changement de configuration est réalisé par des opérations de reconfiguration bien identifiées dont l'influence sur le comportement temporel du système est analysée à travers les capacités de prise en charge de ces activités dans l'ordonnancement. L'approche développée propose de modéliser le comportement adaptatif à travers la définition du contexte, de la variabilité, des opérations de reconfigurations et de la configuration de base. L'analyse d'ordonnançabilité est ensuite effectuée au niveau du modèle en intégrant l'impact des comportements d'adaptation. Deux paradigmes de modélisation peuvent alors être exploités pour effectuer cette analyse : les requêtes et les flots de données. Enfin, l'approche permet de générer des implantations des comportements adaptatifs à partir des modèles afin d'automatiser l'intégration des mécanismes d'adaptation dans les systèmes temps réel.

Afin de faciliter l'utilisation de l'approche proposée, nous avons implémenté deux solutions de création du comportement adaptatif, qui permettent de générer partiellement ou complètement ce comportement, en vue de faciliter sa création et sa modification. Les deux solutions sont évaluées en termes de nombre des éléments UML créés manuellement ou générés. D'autre part, l'analyse d'ordonnancement est réalisée en utilisant un outil basé sur le formalisme d'UML et le profil MARTE. Cela permet de vérifier que les contraintes temporelles de notre système resteront satisfaites en intégrant les opérations de reconfiguration issues du comportement adaptatif.

Mots-clé : Adaptabilité, adaptation, variabilité, reconfiguration, systèmes temps réel, Ingénierie dirigée par les modèles, UML, MARTE.

Abstract

Real-time systems can be large, distributed and have a dynamic environment. This requires the introduction of various operating modes and reliability techniques. Different operating modes are associated with a different architecture and behavior. Dynamic changes between these modes have an impact on the temporal characteristics of systems which requires an analysis whether the constraints of the system are also fulfilled during adaptations. The adaptation behavior amplifies the complexity of the development of real-time systems. The work presented in this thesis is focused on specifying the adaptability and the study of the temporal impact of dynamic adaptation actions on a predefined set of configurations.

For this purpose, we present a tooled methodology based on the concept of Mode (extension of UML state machine) of the MARTE profile. Each mode represents a possible behavior of the system for a well determined environment associated with a software configuration. The configuration change is performed by a clearly identified reconfiguration operation. The influence of these operations on the temporal behavior of the system is done via schedulability analysis. This methodology proposes to model the adaptive behavior through the definition of the context, the variability, the reconfiguration operations and of the base configuration. The schedulability analysis is performed at the model level by incorporating the impact of the behavior of adaptation. Two paradigms of modeling can be exploited to perform this analysis : request/reply and data flow. Finally, the approach allows generating the implementation of adaptive behavior from the model to automate the integration of adaptation mechanisms in real-time systems.

In order to facilitate the use of the proposed approach, we implemented two solutions to create adaptive behavior, which allow to generate this behavior completely or partially, to facilitate its creation and modification. Both solutions are evaluated in terms of the number of UML elements created manually or generated. Furthermore, the schedulability analysis is performed using a tool based on the formalism of UML and the MARTE profile. This allows to verify that the temporal constraints of our system will remain satisfied even with the inclusion of reconfiguration operations executing the adaptive behavior.

Keywords : Adaptability, adaptation, variability, reconfiguration, real time systems, Model Driven Engineering, UML, MARTE.

إلى والدَيْ العزيزين ‹رشيد بوخنوفة› و ‹بيبية بن لطرش›

A mes chers parents « Rachid Boukhanoufa » et « Bibia Benlatreche »

إن يحسن العلم ينفع وحده مالم يتوج ربه بخلق حافظ إبراهيم*

---

*Verse of the poet **Hafedh Ibrahim:**
Never think that science alone is a benefit, unless the owner is crowned by the morality

*Vers du poète **Hafedh Ibrahim :**
Ne pensez jamais que seule la science est un avantage, à moins que son propriétaire soit couronné par la morale.

Remerciements

J'adresse tout d'abord mes remerciements à François TERRIER, directeur de thèse et chef du Laboratoire d'Ingénierie dirigée par les modèles pour les Systèmes Embarqués (LISE) au CEA LIST, de m'avoir accueilli au sein de son laboratoire ainsi que pour avoir conduit et guidé mes travaux.

Je remercie Ansgar RADERMACHER pour avoir encadré cette thèse, ces conseils précieux m'ont permis d'améliorer mes connaissances et d'aboutir à la production de ce travail. Je voudrais aussi le remercier pour le temps qu'il m'a accordé tout au long de ces années.

Mes plus sincères remerciements vont également aux membres du jury, qui ont accepté d'évaluer mon travail de thèse. Merci à M. Laurent PAUTET et M. Jean-Michel BRUEL d'avoir accepté d'être les rapporteurs de ce manuscrit, de m'avoir fait l'honneur de juger ce travail avec intérêt. Merci également à M. Alain MERIGOT et M. Lionel SEINTURIER d'avoir accepté d'examiner ce travail et de faire partie de mon jury de thèse. À tout le jury, j'exprime ma reconnaissance pour les remarques, les questions et les recommandations qui ont été formulées.

Un grand remerciement est également adressé à tous les membres du laboratoire LISE. Je remercie tout particulièrement : Arnaud, Sara et Chokri pour les discussions toujours utiles qui ont créé un cadre de travail convivial et collaboratif. Tous mes collègues ainsi que les doctorants du LISE pour leur soutien amical et leurs encouragements tout au long de cette thèse. Cette atmosphère m'a permis d'apprécier toutes ces années et de me sentir au sein d'un milieu familial.

Je profite également pour remercier mon entourage pour m'avoir soutenu durant ces années, en particulier, Abderraouf Benyahia, Kaddour Benyahia, Yasmina Houhou, Lazhar Saidani, Rafik Benhiza, Younes Behloul, Mostafa Abdelaziz, Mohamed El-Amine Brahmia, Mohamed Guennoun et Mohamed-Tayeb Boudra.

De plus, mes remerciements seraient incomplets, si je ne fais pas mention des personnes les plus cher à mes yeux, qui ont pu supporter mon éloignement et ont pu continuer à me soutenir pour mes études. Je vous adresse mes chaleureux remerciements, Rachid, Bibia, Abdellah, Zineb, Youcef, Daoud, Idris et Rahma.

Enfin, que tous ceux qui m'ont soutenu, encouragé, conseillé, trouvent ici l'expression de ma profonde gratitude.

Antony, le 9 septembre 2012.

Table des matières

CHAPITRE 1

Introduction générale

1.1 Contexte et problématique

Le développement des systèmes temps réel embarqués (STRE) pose des défis importants pour les développeurs. Dans de tels systèmes, l'exactitude des résultats ne dépend pas seulement de l'exactitude logique des calculs, mais aussi de la date à laquelle le résultat est produit. Cela implique que le temps de réponse est aussi important que la production des résultats corrects. C'est pourquoi, les systèmes temps réel doivent être conçus d'une manière permettant de respecter ces temps de réponse lors de l'exécution. Plusieurs solutions ont été proposées dans différents cadres tels que la conception multi-tâches, la conception synchrone ou plus récemment dans le cadre de l'ingénierie dirigée par les modèles (IDM) pour surmonter ces difficultés. Cependant, le changement constant qui caractérise les environnements de ces systèmes (comme : l'état de la batterie, une panne matérielle partielle, le changement de position, de nouvelles exigences, mises à jour disponibles, etc.) constitue un défi supplémentaire pour les développeurs. Afin de conserver la convivialité (usability) et la fiabilité des systèmes, ils doivent être adaptés aux changements de leur environnement opérationnel. Cette adaptation est faite en changeant l'architecture et le comportement du système de manière dynamique. Cela exige la mise en place de différents modes de fonctionnement et techniques de fiabilité. De plus, ces différents changements dynamiques d'architecture et de comportement ont un impact sur les caractéristiques temporelles des systèmes qui nécessitent une étude particulière des capacités à garantir les contraintes fixées aux systèmes. Autrement dit, les opérations de reconfiguration dynamique peuvent influencer le temps de réponse du système et peuvent alors entraîner des violations des échéances. Par ailleurs, ces comportements d'adaptation amplifient la complexité du développement des systèmes temps réel. Autrement dit, la modélisation du comportement adaptatif (création et modification) doit être facile et doit permettre l'analyse d'ordonnançabilité.

Dans les systèmes temps réel à contraintes dures une analyse d'ordonnançabilité dans la phase de développement est nécessaire. En ce qui concerne les systèmes adaptables, cela nécessite d'avoir un modèle complet du comportement adaptatif. Cependant, ce modèle peut être d'une grande taille ce qui rend sa création et modification difficile et source d'erreurs. La taille du modèle de comportement adaptatif peut être grande, compte tenu de l'explosion du nombre de configurations possible lié au nombre de variantes possibles pour chaque composant dans le système. Bien que l'IDM soit une approche de développement attractive, les efforts nécessaires pour créer ou réviser le modèle du comportement adaptatif restent souvent source d'erreurs et de difficulté.

Les systèmes non adaptatifs sont fondés principalement sur une configuration unique sur laquelle une analyse d'ordonnançabilité est appliquée. En revanche, les systèmes adaptatifs sont basés principalement sur plusieurs configurations. Chaque configuration est analysée séparément. Cette analyse n'est pas suffisante, vu qu'elle ne prend pas en compte la charge supplémentaire des opérations de reconfiguration. Cette charge supplémentaire influence souvent le fonctionnement du système de point de vue de ses caractéristiques temporelles et doit être intégrée dans la démarche de l'ingénierie des modèles.

1.2 Proposition

Le travail présenté dans cette thèse est focalisé sur la spécification de l'adaptabilité d'un système temps réel et l'étude sur des jeux de configurations prédéfinis de l'impact temporel des actions d'adaptation dynamique.

Pour cela, nous présentons une méthodologie outillée basée sur la notion de Mode (extension de machine à état UML) du profil MARTE. Chaque mode représente un comportement possible du système pour un environnement bien déterminé associé à une configuration logicielle. Pour faciliter la description des architectures liées aux différents modes, l'application est modélisée au moyen d'une approche par composant. Cela permet d'utiliser la notion de connecteur et de gérer le changement des connexions entre composants durant l'exécution. Le changement de configuration est réalisé par des opérations de reconfiguration dédiées et explicites. Il devient alors possible d'analyser l'influence de ces opérations sur le comportement temporel du système, à travers les capacités de prise en charge de ces activités dans l'ordonnancement. La modélisation du comportement adaptatif s'effectue alors sur quatre plans : les modes de fonctionnement, le contexte, la variabilité et les opérations de reconfiguration. L'analyse d'ordonnançabilité est ensuite effectuée au niveau du modèle en intégrant l'impact des comportements d'adaptation. Deux paradigmes de modélisation peuvent alors être exploités pour effectuer cette analyse : les requêtes et les flots de données. Le paradigme de modélisation basé sur la communication par requête permet de modéliser tous types de système. Cependant, il offre moins de facilité pour l'analyse d'ordonnançabilité, cf. fin de section 2.3.2 pour plus de détail. En revanche, le paradigme de communication par flot de données est plus avantageux pour l'analyse d'ordonnançabilité, mais ne permet pas de modéliser tous types de système. Enfin, l'approche permet de générer des implantations des comportements adaptatifs à partir des modèles afin d'automatiser l'intégration des mécanismes d'adaptation dans les systèmes temps réel. Dans l'objectif de faciliter l'utilisation de l'approche proposée et son évaluation, nous avons implémenté des outils de création du comportement adaptatif, qui permettent de générer partiellement ou complètement ce comportement, en vue de faciliter sa création et sa modification. Les solutions sont évaluées en termes de nombre des éléments UML créés manuellement ou générés. La validation des solutions est réalisée par une analyse d'ordonnancement qui utilise un outil basé sur le formalisme d'UML et le profil MARTE. Cela permet de vérifier que les contraintes temporelles de notre système resteront satisfaites en intégrant les opérations de reconfiguration issues du comportement adaptatif.

1.3 Plan de thèse

Le manuscrit de cette thèse se compose d'un chapitre qui décrit le contexte, la problématique et les objectifs et cinq chapitres suivis d'une conclusion et des perspectives.

Le deuxième chapitre présente les concepts de bases et les terminologies liées aux systèmes temps réel adaptables nécessaires à la compréhension de la suite de cette thèse. Notamment l'ingénierie dirigée par les modèles, UML et le profil MARTE. Ensuite, nous présentons des notions de base pour les systèmes temps-réel embarqués. Dans le dernier point de ce chapitre, nous présentons la définition de l'adaptabilité, ses catégories et ses

niveaux d'application.

Dans le troisième chapitre, nous présentons les critères de comparaison des travaux connexes (related work), la description et la classification de ces travaux et leurs discussions. Ces travaux seront présentés selon les méthodologies de développement logiciel et les outils qui adressent :

- Le développement des systèmes temps réel dans le cadre de l'ingénierie dirigée par les modèles
- Le développement des systèmes adaptatifs dans le cadre de l'ingénierie dirigée par les modèles
- Le développement des systèmes temps réel adaptable de manière générale.

Le quatrième et le cinquième chapitre montrent la méthodologie de conception et l'outillage que nous avons mis en place pour la modélisation du comportement adaptatif et l'analyse d'ordonnançabilité avec la prise en compte de la charge temporelle des opérations de reconfiguration liées au comportement adaptatif du système.

Le sixième chapitre présente une évaluation de notre méthode sur un exemple robotique et démontre les avantages de notre méthodologie.

Enfin la conclusion retrace nos travaux effectués, présente quelques améliorations qui pourraient être apportées et ouvre la voie sur une poursuite des travaux.

CHAPITRE 2

Contexte

2.1 Introduction

Dans ce chapitre nous allons présenter le contexte de travail de cette thèse qui étudie le développement des systèmes temps réel adaptables dans le cadre de l'ingénierie dirigée par les modèles. Dans ce cadre, nous présentons la définition de l'adaptabilité des systèmes, ses catégories et ses niveaux, ainsi que les types de changement possible sur une configuration système. Nous présentons également des notions de base des systèmes temps réel et de l'analyse d'ordonnançabilité pour deux paradigmes de modélisation (*requête*, *DataFlow*) qui ont un lien avec l'adaptabilité. Ensuite nous présentons l'ingénierie dirigée par les modèles, *UML* et le profil *MARTE* que nous allons utiliser pour la modélisation. Dans le dernier point de ce chapitre, nous effectuons une analyse des besoins pour le travail de cette thèse. Tous ces éléments sont liés au développement des systèmes temps réel adaptables dans le cadre de l'ingénierie dirigée par les modèles. Enfin nous concluons le chapitre.

2.2 L'adaptabilité des systèmes

Dans cette section, nous présentons quelques définitions de l'adaptabilité des systèmes. Ensuite, nous définissons les éléments de l'adaptabilité : l'environnement du système, la configuration du système et ses types de changement. Ainsi que les catégories d'adaptabilité.

2.2.1 Définition de l'adaptabilité

Il existe plusieurs définitions dans la littérature de l'adaptabilité et de l'adaptation. Nous retiendrons celle proposée dans [1] et utilisée dans [2] et [3] où l'adaptation signifie le changement du système afin d'accommoder le changement de son environnement. Plus exactement, cette définition considère que l'adaptation d'un système logiciel (S) est causée par le changement (ChE) d'un ancien environnement (E) à un nouvel environnement (E') et aboutit à un nouveau système (S') qui répond idéalement aux besoins de son nouvel environnement (E'). Formellement l'adaptation peut être considérée comme une fonction : $Adaptation : E \times E' \times S \rightarrow S'$, où $meet(S', need(E'))$.

De cette façon, un système est adaptable si une fonction d'adaptation existe. D'une manière générale, l'adaptabilité se réfère alors à la capacité du système pour exécuter l'adaptation.

En effet, l'adaptation implique trois tâches :
- La capacité de reconnaître/détecter le changement ChE dans l'environnement
- La capacité de déterminer le changement ChS à faire au système S en fonction du changement ChE
- La capacité d'effectuer le changement pour produire le nouveau système S'.

Ils peuvent être écrits comme des fonctions de la façon suivante :
- $EvenChangeRecognition : E' \times E \rightarrow ChE$
- $SysChangeRecognition : ChE \times S \rightarrow ChS$
- $SysChange : ChS \times S \rightarrow S'$, où $meet(S', need(E'))$ est vérifiée.

La fonction *meet* confirme que le nouveau système S' répond en effet aux besoins (*need*) du nouvel environnement E'.

D'autres définitions sont proposées dans la littérature, par exemple [4] définit *l'adaptation logicielle* (*software adaptation*) comme n'importe quelle modification logicielle qui change la fiabilité ou la ponctualité (*timeliness*) du logiciel sans affecter d'autres aspects de sa fonctionnalité. Les adaptations sont décrites par des *algorithmes d'adaptation*. Les algorithmes d'adaptation décrivent les changements à exécuter et les circonstances qui déclenchent l'adaptation. L'adaptation logicielle englobe beaucoup de techniques du réglage logiciel (*software-tuning*). Il s'agit notamment de :

- L'allocation des ressources, telles que le déplacement d'un composant logiciel d'un processeur à un autre,
- Ajustements de l'ordonnancement du processeur,
- Modification des facteurs de réplication pour N composants logiciels modulaires redondants, et,
- Modification des échéances ou la période pour la livraison d'un service par un composant logiciel.

Dans les systèmes temps réel, ces adaptations peuvent être entrelacées. Par exemple, la modification d'une échéance d'une tâche peut forcer des rajustements dans l'ordonnancement du processeur.

Dans notre travail nous optons pour la première définition "*l'adaptation d'un système logiciel (S) est causée par le changement (ChE) d'un ancien environnement (E) à un nouvel environnement (E′) et aboutit à un nouveau système (S′) qui répond idéalement aux besoins de son nouvel environnement (E′)*", parce qu'elle est plus générale, alors que la deuxième définition n'autorise pas les changements des fonctionnalités. Pour pouvoir utiliser la première définition nous devons délimiter l'environnement, les catégories de l'adaptation et tous les types de changement que nous pouvons faire sur le système.

2.2.2 Le système et l'environnement

Une *application* est composée d'un *système informatique* et d'un *environnement* physique avec lequel il interagit. Un système est une entité qui interagit avec d'autres systèmes matériels, logiciels ou humains et le monde physique. Ces autres systèmes constituent l'environnement du système étudié. La frontière du système est la limite commune entre le système et son environnement (cf. figure 2.1).

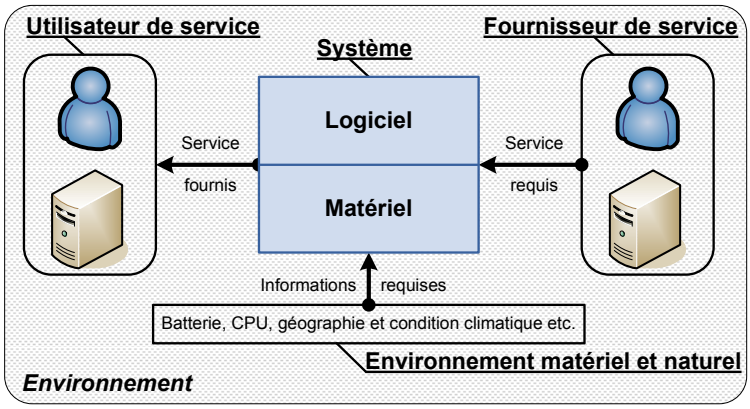


FIGURE 2.1 – Système et environnement.

Les services fournis par un système sont le comportement perçu par son ou ses utilisateurs. L'utilisateur de service est un autre système qui bénéficie d'un service d'un système. Le service requis par le système représente des éléments qui peuvent influencer son comportement. Le fournisseur de service à un système est un autre système. Un système implémente généralement plus d'une fonction, et délivre plus d'un service.

2.2.3 Une configuration

Une configuration d'un système peut être vue comme un ensemble de composants interconnectés en vue d'interagir. Chaque composant peut être hiérarchique et sa décomposition donne un nouveau sous-système. La décomposition s'arrête quand un composant est considéré comme un composant atomique. Nous définissons la configuration d'un système comme la description de l'ensemble des composants et de leurs interconnexions.

En fait, un cycle de l'adaptation consiste à détecter un changement dans l'environnement, calculer le changement à faire dans le système et en suite effectuer le changement sur le système. Les changements possibles dans le système sont de type de : structure, interface, déploiement ou implémentation.

2.2.4 Les types de changement dans un système

– **Les changements de structure** modifient la configuration structurelle d'un système. Les changements structuraux de base sont : l'ajout, la suppression de composant et la modification des interconnexions entre composants. Sur la figure 2.2 le changement de structure effectué est une modification de la connexion $Con1$ pour rediriger les requêtes en provenance du composant $C3$ vers le composant $C5$. ***Dans cette thèse, nous prenons en compte le changement de structure dans la phase de modélisation.***

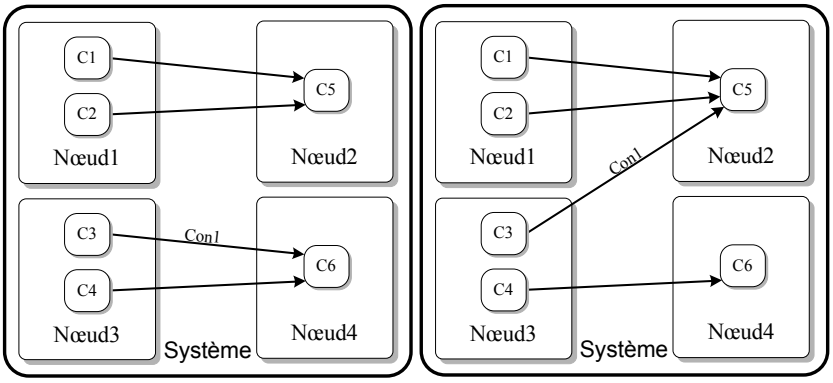


FIGURE 2.2 – Un exemple de changement de structure : redirection de la connexion $Con1$ vers le composant $C5$.

– **Les changements de déploiement** modifient uniquement le placement des composants en termes de nœud de calcul (calculateur). Ce type de changement ressemble au problème de la migration de processus, si l'on rapproche un composant à un processus. La figure 2.3 illustre un exemple de changement géométrique, dont le composant $C6$ qui s'exécutait sur le calculateur 2 va continuer son exécution sur le calculateur 3 sans perte de cohérence pour l'application. Ce type de changement ne modifie pas

la configuration structurelle de l'application, c'est-à-dire, il n'y a aucune modification de la structure logique (l'architecture logicielle) de l'application et de la topologie des interconnexions.

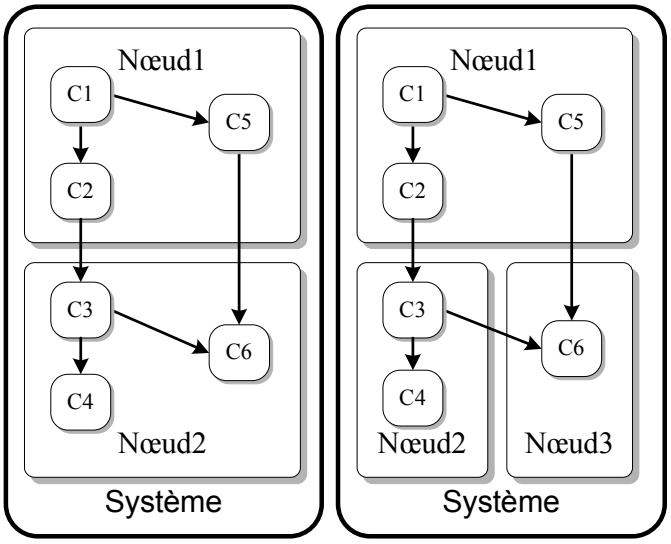


FIGURE 2.3 – Un exemple du changement de déploiement : le composant $C6$ est déplacé du $Noeud2$ vers le $Noeud3$

– **Le changement d'interface** est défini comme la modification de l'ensemble des services fournis par un composant (cf. figure 2.4 B) ainsi que la modification de la signature d'un service (cf. figure 2.4 A). La signature d'un service correspond à son nom, à son nombre de paramètres et à leur type. Dans la figure 2.4 A, nous pouvons remarquer que la signature de la méthode P a changé, le nombre de paramètres a été modifié. Dans la figure 2.4 B, un service a été ajouté à l'ensemble des services fournis par le composant $C3$. Les changements d'interface posent des problèmes de gestion de version. En effet, les clients d'un composant ne doivent pas être perturbés par le changement d'interface des services du composant.

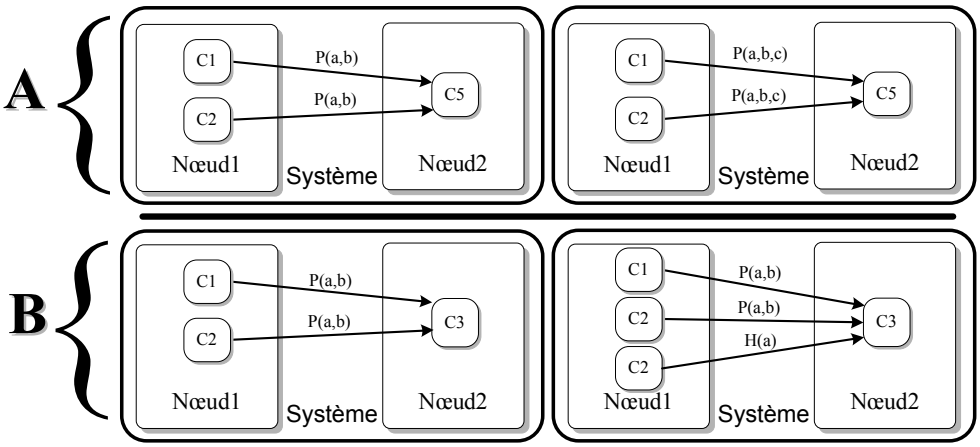


FIGURE 2.4 – Un exemple du changement d'interface

– **Le changement d'implémentation** modifie le code des composants et éventuellement

restructure leurs données sans modifier leurs interfaces. Par contre il ne change ni la structure de l'application ni le placement, ni l'interconnexion des composants.

Des techniques de reconfiguration dynamique sont définis afin de réaliser ces changements sur un système en cours d'exécution sans avoir à l'arrêter complètement. Généralement ces techniques sont basées sur la notion d'état sûr (*safe state*) de composant [5, 6, 7]. L'idée générale de l'état sûr d'un composant consiste à compter le nombre de *thread* en cours d'exécution pour le compte de ce composant. Une fois le nombre de *thread* est égal à zéro, le composant est en état sûr. Une fois le composant est en état sûr, toutes les opérations de changement sont possibles avec la garantie qu'ils ne vont pas affecter la cohérence du composant [8, 9, 10].

Les techniques de reconfiguration peuvent toucher tous les niveaux de la couche logicielle d'un système informatique. Ces opérations sont possibles dans les applications [6, 7], les middlewares [11] et les systèmes d'exploitation [8, 5]. ***Dans le cadre de cette thèse, notre travail est autour de l'adaptation des applications que nous appelons désormais système.***

2.2.5 Les catégories d'adaptabilité

Tel que précisé précédemment, le calcul du changement à faire dans le système pour suivre le changement de l'environnement est une tâche parmi les tâches de l'adaptation. Le résultat de cette tâche représente un plan de reconfiguration. Le plan de reconfiguration détermine les changements à faire dans le système en détail. Par exemple, sur la figure 2.3 le composant $C6$ est déplacé du calculateur $Noeud2$ vers le calculateur $Noeud3$, cette information sera ici représentée dans le plan de reconfiguration par l'opération $move(C6, Noeud2, Noeud3)$.

Le calcul du plan de reconfiguration représente un point de classification de l'adaptabilité.

Le travail présenté dans [12, 10] classifie l'adaptabilité en trois catégories selon la méthode de calcul du plan de reconfiguration : *constructible plan*, *predefined plan* et *Intelligent plan*.

– **Le plan constructible** (*constructible plan*) : cette catégorie se rapporte à des systèmes combinant un *langage de description* pour décrire la configuration initiale ; un *langage de modification* pour programmer le plan de reconfiguration ; et le *gestionnaire de reconfiguration*. Le langage de modification supporte typiquement l'addition et le déplacement des éléments du système (comme des composants ou des liens), l'activation, la désactivation et la migration de ces éléments. Quand le changement doit se produire, un programmeur développe une configuration alternative qui résout le problème. Dans certains cas, cela peut être difficile et le programmeur doit utiliser plusieurs itérations, où les candidats (configurations possibles) sont évalués pour trouver la solution définitive. Le programmeur implémente les changements en envoyant des directives au gestionnaire de configuration, qui interprète et exécute ces directives (cf. figure 2.5 A).

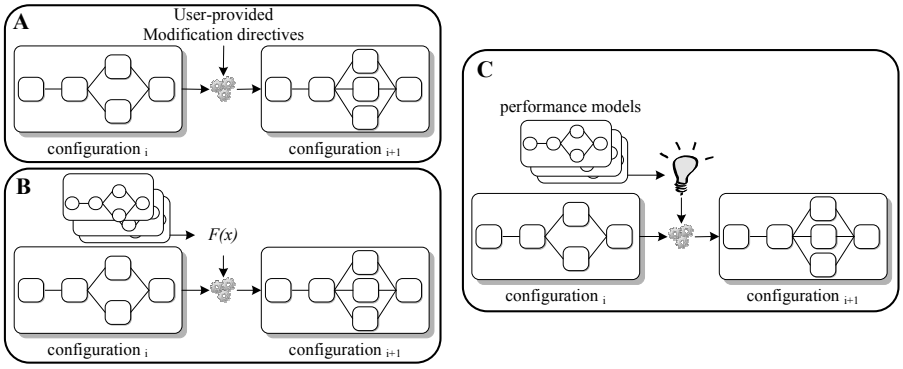


FIGURE 2.5 – Catégories d'adaptabilité : (A) *constructible plan*, (B) *predefined plan* et (C) *Intelligent plan* (source [10])

- **Le plan prédéfini** (*predefined plan*) : est basé sur un jeu de configurations prédéfinies, conçu et vérifié statiquement (offline). A l'exécution, le système peut sélectionner une alternative de celles disponibles, en fondant la décision sur une fonction d'aptitude connue, et exécutant le plan de reconfiguration correspondant. Pour chaque configuration définie, le système doit connaître ses plans de reconfiguration vers toutes les autres configurations (cf. figure 2.5 B).
- **Le plan intelligent** (*Intelligent plan*) : dans cette catégorie, contrairement à ce qui précède, une automatisation enlève la restriction de l'ensemble limitée de configuration. Dans ce cas, un gestionnaire d'adaptation intelligent calcule le plan de reconfiguration approprié en utilisant des modèles de performance (cf. figure 2.5 C). Ce cas n'est pas favori pour les systèmes temps réel stricts compte tenu de l'analyse nécessaire en phase d'exécution pour ce type de système. ***Dans le cadre de cette thèse, notre travail est autour de l'adaptabilité avec des plans de reconfiguration prédéfinis.***

Dans cette thèse nous travaillons sur les systèmes temps réel, notamment la modélisation des caractéristiques temporelles et leur analyse. Dans ce qui suit nous présentons une définition des systèmes temps réel, leurs catégories et la nature de leurs tâches. Ensuite nous présentons l'analyse d'ordonnançabilité pour le paradigme de modélisation par flot de données sur lesquels notre solution est basée (cf. chapitre 5).

2.3 Les systèmes temps réel

A l'égard de certains systèmes, la validité d'une action dépend du temps qui s'écoule entre la fin de la réalisation de cette action et l'événement qui l'a déclenchée. Autrement dit, sur ce type de systèmes, il est nécessaire que les réactions du système aux événements soient effectuées en un temps inférieur à une borne maximale. Les applications soumises à ce type de contraintes sont qualifiées de *temps réel* et les bornes maximales sur le temps de réalisation des actions sont appelées *contraintes temps réel*.

Définition 1 *Un système temps réel est un système dont l'exactitude des résultats ne dépend pas seulement de l'exactitude logique des calculs mais aussi de la date à laquelle le résultat est produit. Si les contraintes temporelles ne sont pas satisfaites, on dit qu'une défaillance système s'est produite. [13]*

Les interactions d'un système temps réel avec son environnement peuvent avoir lieu à des moments déterminés par une référence de temps interne au système (on parle alors de système temps réel basé sur l'horloge -temps-) ou à des moments déterminés par l'environnement lui-même (on parle alors de système temps réel basé sur les événements).

Dans le cadre des systèmes temps réel, le non-respect d'une contrainte temporelle peut avoir des conséquences plus ou moins catastrophiques en termes de pertes humaines, écologique ou économique, etc. Par exemple, lors de l'atterrissage d'un avion, le système d'indication d'altitude doit fournir une valeur exacte à un instant précis de manière à ce que les réactions du pilote, qui dépendent de l'altitude, aient lieu au bon moment. Tout retard de réponse du système peut conduire dans ce cas à l'écrasement de l'avion. C'est ainsi le cas dans une application de robot mobile autonome capable de détecter des obstacles ; la détection tardive d'un obstacle se trouvant sur sa trajectoire peut ici se terminer par une collision capable d'endommager le robot.

Selon la nature de l'application, il est possible de tolérer de temps en temps et avec une certaine marge le non-respect de certaines contraintes. Ce type de contraintes est appelé *contraintes relatives/souples* en opposition aux contraintes dites *strictes/dures* qui doivent impérativement être respectées.

2.3.1 Les catégories des systèmes temps réel

Ainsi, suivant les contraintes temporelles d'un système réactif, on peut distinguer deux grandes catégories des systèmes temps réel :

- **Systèmes temps réel à contraintes strictes/dures (*hard real time constraints*) :** lorsque toutes les contraintes temporelles doivent être impérativement respectées. Dans ce cas la fin d'exécution d'une action après l'échéance (*deadline*) déclenchera une exception. Le non-respect des contraintes peut provoquer des conséquences catastrophiques [14]. Les systèmes de contrôle de vol, systèmes de contrôle de station nucléaire, systèmes de contrôle de voies ferrées en sont des exemples. ***Le travail présenté dans ce mémoire est autour des systèmes temps réel à contraintes strictes.***
- **Systèmes temps réel à contraintes relatives/souples (*soft real time constraints*) :** au contraire des systèmes durs, dans ce cas la fin d'exécution d'une action après l'échéance (*deadline*) ne déclenchera pas une exception. En effet, le non-respect des contraintes temporelles est toléré (acceptable) par le système, et sans que cela ait des conséquences catastrophiques [15, 16]. Par exemple des applications multimédias.
- **Systèmes temps réel à contraintes mixtes :** sont composés des tâches à contraintes strictes et des tâches à contraintes relatives [17].

Un programme temps réel est composé d'un ensemble d'entités appelées *tâches temps réel*. Chacune ayant un rôle qui lui est propre, comme par exemple : réaliser un calcul, être associé à une alarme, traiter des entrées / sorties, etc.

La loi d'arrivée d'une tâche définit sa nature. Cette loi s'agit des contraintes temporelles qui définissent la répartition des dates d'activation des instances d'une tâche dans le temps. Selon la loi d'arrivée, il est possible de classer les tâches en trois catégories :

- **Une tâche *périodique*** est une tâche dont l'activation est régulière et le délai P_i (*période*) entre deux activations successives est constant. Une tâche T_i est caractérisée par une durée d'exécution C_i, une période d'activation P_i, une échéance D_i et la date de la

première activation R_i (cf. figure 2.6).

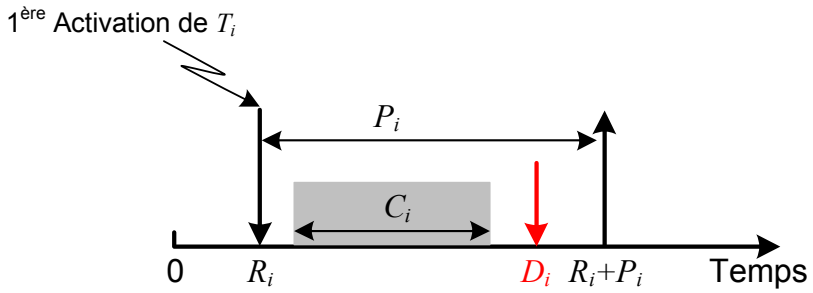


FIGURE 2.6 – Modèle d'une tâche temps réel périodique

- **Une tâche *sporadique*** est une tâche caractérisée par un délai minimum entre deux activations successives. Au contraire des tâches périodiques, les dates d'activation des différentes instances d'une tâche sporadique ne peuvent pas être déterminées a priori.
- **Une tâche *apériodique*** est une tâche dont on ne connaît aucune caractéristique, sauf son échéance. Elle est généralement activée par l'arrivée des évènements (message ou requête de l'opérateur) qui peuvent être produits à tout instant.

Les applications temps réel à contraintes strictes sont des applications où les contraintes temporelles doivent être strictement satisfaites. Dans ce but, une analyse des contraintes temporelles en phase de conception (off-line) est essentielle.

2.3.2 L'analyse d'ordonnançabilité

Comme nous l'avons précisé avant, l'exactitude des résultats des systèmes temps réel embarqués, ne dépend pas seulement de l'exactitude logique des calculs mais aussi de la date à laquelle le résultat est produit. C'est pour quoi, lorsque nous concevons un système temps réel, nous devons nous assurer qu'il répond à ces propriétés [18] :

- **Exactitude de fonctionnement** (*Correctness of functionality*) : par cette propriété, nous nous attendons à ce que notre système produise un résultat correct pour chaque ensemble de données en entrée. Des techniques de vérification traditionnelles comme le test et la preuve formelle peuvent être utilisées pour démontrer l'exactitude fonctionnelle.
- **Exactitude du comportement temporel** (*Correctness of timing behavior*) : les exigences d'un système temps réel incluent les propriétés temporelles qui doivent être respectées par l'implémentation. Les échéances (*deadlines*) peuvent être assignées aux fonctions particulières du système et ensuite aux tâches qui implémentent ces fonctions. Le comportement temporel est vérifié en vérifiant que les temps d'exécution des tâches ne dépasseront jamais les échéances requises. Cette analyse du comportement temporel est appelé *analyse d'ordonnançabilité*.

Pour pouvoir s'assurer du respect des contraintes temporelles lors de l'exécution du système, en particulier quand il s'agit d'un système temps réel strict, une analyse de l'ordonnancement doit être effectuée en utilisant des analyseurs d'ordonnançabilité en phase de conception, parmi lesquels on peut citer Cheddar [19], MAST [20], RT-Druid [21], Optimum [22], etc. Les analyseurs cité ci-dessus sont tous basé sur la notion de tâche dans leur modèle d'analyse.

En ce qui concerne la couche utilisée pour réaliser l'analyse d'ordonnançabilité c'est la couche tâche. Cependant, la conception (modélisation) d'une application n'est pas faite dans la couche tâche. En modélisant, nous utilisons un paradigme (cf. 2.7). Selon le paradigme, il faut faire le passage vers un modèle à base de tâches temps réel pour pouvoir réaliser l'analyse d'ordonnançabilité. Le paradigme le plus proche au modèle de tâche et le plus utilisé est bien le paradigme de flot de données synchrone (SDF *Synchronous Data Flow*) [23], qui représente un cas particulier du paradigme *Data Flow*.

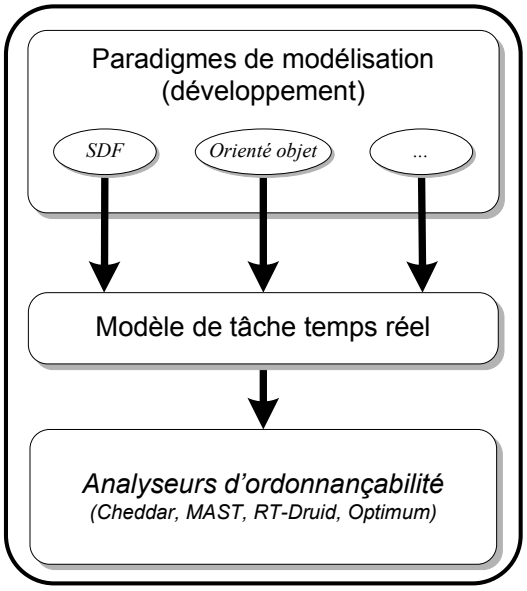


FIGURE 2.7 – Paradigmes de modélisation et l'analyse d'ordonnançabilité.

Le flot de données (*Data Flow*)

Un programme/modèle de flot de données (*data flow program/model*) [24] est un graphe orienté dans lequel chaque nœud (*acteur*) représente une fonction et chaque arc orienté un support conceptuel sur lequel les éléments (*jetons*) de données coulent, comme illustré dans la figure 2.8. Les cercles à fond blanc représentent les nœuds, les flèches représentent les flux et le cercle plein représente un jeton. Un modèle de flot de données peut être hiérarchisé, étant donné qu'un nœud peut représenter un graphe de flot de données.

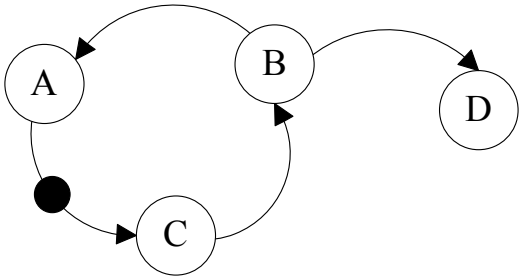


FIGURE 2.8 – Graphe de flot de données.

Le comportement d'un processus de flot de données est une séquence de déclenchement

(*firings*). Pour chaque déclenchement, des jetons sont consommés et des jetons sont produits. Le nombre de jetons consommés et produits peut varier pour chaque déclenchement et il est défini dans les règles de déclenchement du flot de données. Les modèles de flot de données sont très utilisés dans les applications de traitement du signal numériques. Le but principal dans les processus de conception basés sur le flot de données est de trouver un ordonnancement statique de déclenchement afin d'utiliser efficacement le paradigme de flot de données sur des ressources partagées. Cependant, pour les modèles de flot de données générale, il est indécidable si un tel ordonnancement existe [24].

Le flot de données synchrone (*SDF : Synchronous data flow*) [25, 26] met des restrictions complémentaires sur les modèle de flot de données, vu que le flot de données synchrone exige que le processus consomme et produit un nombre fixe de jetons pour chaque déclenchement. Avec cette restriction, il est garanti qu'un ordonnancement statique peut toujours être trouvé. La figure 2.9 montre un flot de données synchrone. Les nombres sur les arcs montrent combien de jetons sont produits et consommés au cours de chaque déclenchement. Un ordonnancement possible pour le flot de données synchrone donné dans la figure 2.9 est A, A, C, C, B, D.

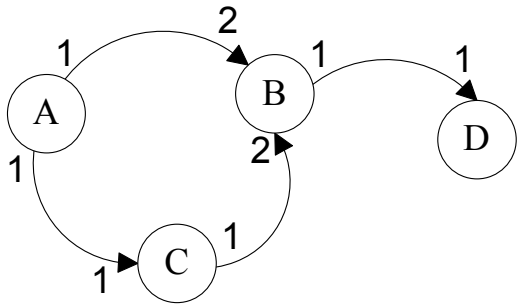


FIGURE 2.9 – Graphe de flot de données synchrone, *SDF*.

Il existe différents modèles de flot de données, pour un excellent aperçu voir [27]. Pour plus d'information sur l'analyse d'ordonnançabilité à base de flot de données voir [23].

Appel d'opération

Dans un paradigme de modélisation orienté objet, la composition (la communication entre les composants applicatif) du système est modélisée par des requêtes (appel d'opération ou envoie de message). Ces appels d'opérations peuvent être synchrones ou asynchrones. L'analyse d'ordonnançabilité des systèmes modélisés avec ce paradigme est difficile [28]. En effet, le passage d'un modèle de ce paradigme vers un modèle de tâche temps réel, toute en respectant le modèle d'exécution du paradigme orienté objet, fournit un modèle difficile à analyser. La raison en est que chaque tâche fait partie de plusieurs *end-to-end computations* et contraintes temporelles, contrairement aux modèles de tâches utilisés dans les travaux de recherche sur l'ordonnançabilité temps réel [28]. Le modèle de calcul influence l'analyse d'ordonnançabilité, toute en la rendant plus facile ou plus difficile.

2.4 L'ingénierie dirigée par les modèles

L'évolution du génie logiciel pour suivre la complexité croissante des logiciels, assurer la séparation des préoccupations (des métiers) et gérer la multiplicité des plateformes (Java, J2EE, Web Services, ...), a amené le passage des technologies/paradigmes procéduraux vers des technologies/paradigmes des modèles, donnant naissance à l'ingénierie dirigée par les modèles.

L'*Ingénierie Dirigée par les Modèles* (IDM), ou *Model Driven Engineering* (MDE) en anglais, a permis plusieurs améliorations significatives dans le développement des systèmes en permettant de se concentrer sur une préoccupation plus abstraite que la programmation classique, suivant le principe de : *"tout est un modèle"* [29].

Par ailleurs, l'ingénierie dirigée par les modèles offre un cadre méthodologique et technologique qui permet d'unifier différentes façons de faire dans un processus homogène. Il est ainsi possible d'utiliser la technologie la mieux adaptée à chacune des étapes du développement, tout en ayant un processus global de développement qui soit unifié dans un paradigme unique [30].

Les initiatives dirigées par les modèles les plus représentatives sont : *Model Driven Architecture* (MDA), *Model Driven Development* (MDD) et *Model Driven Engineering* (MDE), ou en français : l'architecture dirigée par les modèles, le développement dirigé par les modèles et l'ingénierie dirigée par les modèles respectivement.

L'architecture dirigée par les modèles (MDA) telle que définie par l'OMG est basée principalement autour des modèles qui sont exprimés dans un langage de modélisation dont le métamodèle est exprimé en MOF (*MetaObject Facility* [31]). L'OMG préconise également l'UML pour la modélisation.

2.4.1 Le langage de modélisation unifié UML (*Unified Modeling Language*)

Depuis le début des années 1980, de nombreux méthodologies orientée objet (OO) ont émergé, chacune spécifiant ses notations et son processus. Parmi les plus éminents été la méthode *Booch*, *Object Oriented Software Engineering* de Jacobson et l'*Object Modeling Technique* (OMT) de Rumbaugh. La plupart des méthodologies ont partagé les mêmes concepts, mais avec des notations différentes.

Vers la fin de 1995 Grady Booch, James Rumbaugh et Ivar Jacobson ont travaillé pour unifier leurs méthodes. Le résultat était la version 0.9 d'UML, qui a été soumise au milieu de 1996 à l'OMG. Vers le début de 1997, la version 1.0 a été publiée, et vers la fin de 1997, la version 1.1 d'UML a été soumise à l'OMG pour la normalisation (standardisation). Dans la même année OMG a adopté UML, dont l'entretien a été repris par le groupe de travail de révision (RTF : *Revision Task Force*) de l'OMG. La version actuelle de l'UML est 2.5 (2012).

Le langage de modélisation unifié (UML) [32] est un langage de modélisation standard pour visualiser, spécifier, construire et documenter des systèmes. UML est un langage de modélisation à usage général. Cela signifie qu'il peut être utilisé avec la plupart des méthodes orientées objet et à base de composant. En outre, UML peut s'appliquer à de nombreux domaines d'application (ex. la santé, la finance, la télécommunication et l'aérospatiale etc). UML peut également être employé avec les principales plates-

formes d'implémentation (exécution), par exemple *Common Object Request Broker Architecture* (CORBA), *Java 2 Enterprise Edition* (J2EE) et *Microsoft .NET*.

2.4.1.1 Les diagrammes d'UML

Un modèle UML se compose des éléments tels que les packages, les classes et les associations. Les diagrammes UML correspondants à ces éléments sont des représentations graphiques des parties du modèle UML. Plus exactement, les diagrammes UML contiennent des éléments graphiques (nœuds reliés par des relations -arcs-) qui représentent les éléments dans le modèle UML.
Chaque diagramme permet de donner une vue du système. Toutefois, il est impossible de donner une seule représentation graphique complète d'un logiciel, ou de tout autre système complexe. De même qu'il est impossible de représenter entièrement une sculpture à trois dimensions par des photos à deux dimensions. Mais il est possible de donner sur un système des vues partielles, semblables chacune à une photo d'une sculpture, et dont l'union donnera une idée utilisable en pratique.
Le langage de modélisation UML, comporte ainsi treize types de diagrammes représentant autant de vues distinctes pour représenter des concepts particuliers des systèmes. Ils se répartissent en deux grands groupes (cf. figure 2.10) :

- *Diagrammes structurels ou diagrammes statiques (UML Structure) :* Diagramme de classes, diagramme d'objets, diagramme de composants, diagramme de déploiement, diagramme de paquet (*Package*), diagramme de structures composites.
- *Diagrammes comportementaux ou diagrammes dynamiques (UML Behavior) :* Diagramme de cas d'utilisation, diagramme d'activités, diagramme de machine à état, diagramme de séquence, diagramme de communication, diagramme global d'interaction, diagramme de temps.

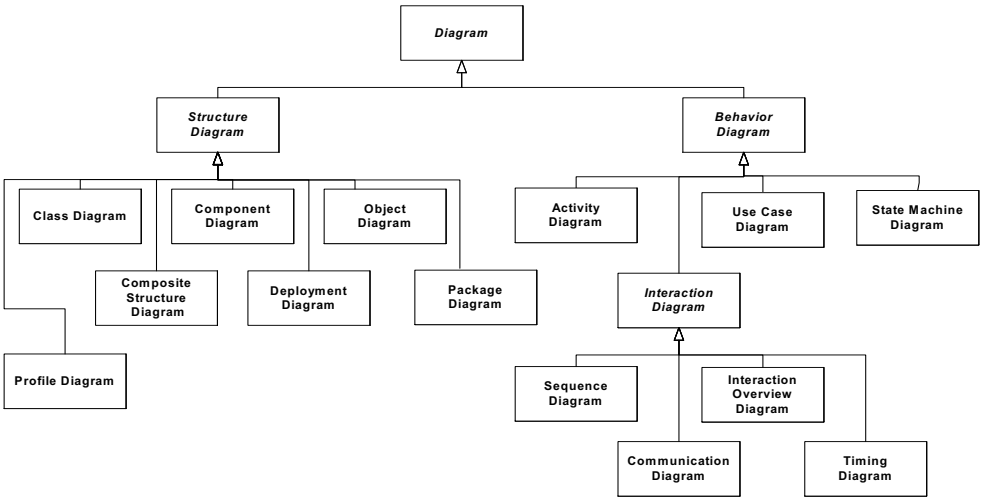


FIGURE 2.10 – La taxonomie des diagrammes de structure et de comportement UML (source OMG [32])

Ces diagrammes sont d'une utilité variable, selon les cas. Ils ne sont pas nécessairement tous utilisés lors d'une modélisation. Les plus utiles sont les diagrammes d'activités, de cas d'utilisation, de classes, d'objets, de séquence et d'états-transitions. Les diagrammes de

composants, de déploiement et de communication sont surtout utiles pour permettre de formaliser les contraintes de réalisation et la solution technique.

Les diagrammes les plus utilisés dans notre contexte de travail sont :

- **Le diagramme de classe :** Le diagramme de classe est une représentation graphique statique des éléments qui composent un système et de leurs relations. Ces éléments représentent les classes du système. La classe est un concept abstrait qui permet de représenter toutes les entités d'un système. Il peut s'agir d'un élément physique (un avion), commercial (une commande) ou logique (une grille de diffusion), de l'élément d'une application (un bouton logiciel d'interface), d'un élément informatique (une table de hachage), comportemental (une tâche) ou tout autre élément devant être modélisé. La classe est définie par son nom, ses attributs et ses opérations.
- **Le diagramme d'activité :** un diagramme d'activité est un graphique de nœuds et de flots qui matérialise le flot de contrôle ou de données, à travers les étapes d'un calcul. L'exécution de ces étapes peut être à la fois simultanée et séquentielle. Une activité présente à la fois des constructions de synchronisation et de branchement. Dans la définition d'une activité, on trouve des nœuds d'activité. Ils représentent l'exécution d'une instruction dans une procédure ou d'une étape dans un enchaînement d'activités et sont connectés par des flots de contrôle et de données. L'exécution d'un nœud d'activité commence à l'endroit où l'on trouve des jetons (indicateurs de contrôle) sur chacun de ses flots d'entrée. Lorsque l'exécution du nœud s'achève, elle se poursuit vers les nœuds qui se trouvent sur ses flots de sortie.
- **Le diagramme de structure composite :** Un diagramme de structure composite est un graphique représentant la structure interne d'une ou plusieurs classes. Une classe sur un diagramme de composite contient un ensemble de parties reliées par des connecteurs. Une partie possède un type et une multiplicité. Les parties peuvent être typées par des composants. Un composant possède un ensemble d'interfaces et une ou plusieurs implémentations.
- **Le diagramme de machine à état :** Un diagramme de machine à état est un graphique représentant des états et des transitions. Il est habituellement relié à une classe et il décrit la réponse d'une instance de la classe aux événements qu'elle reçoit. On peut également rattacher les machines à états à des comportements, des cas d'utilisation et à des collaborations pour décrire leur exécution. Une machine à états est un modèle de tous les états possibles que peut prendre un objet de classe. Elle résume toute influence en provenance du reste du monde comme un événement. Lorsqu'un objet détecte un événement, il y répond en fonction de son état actuel. Cette réponse peut comprendre l'exécution d'un effet et d'un changement vers un autre état.
- **Le diagramme de profil** qui est classé parmi les diagrammes structurels, permet d'étendre le langage de modélisation UML. Le langage UML été conçu pour prendre en charge une grande variété de domaine. Cependant, chaque domaine a des notions particulières, des besoins particuliers, qu'UML ne peut pas couvrir. Ainsi, UML offre un mécanisme d'extensibilité basé sur les profils, qui est une caractéristique fondamentale.

2.4.1.2 Extensibilité d'UML : la notion de profil

Le langage UML propose le mécanisme de profil qui permet aux modélisateurs d'effectuer des extensions courantes sans avoir à modifier le langage de modélisation. Les profils UML apportent un mécanisme permettant de spécialiser UML pour chaque contexte de travail. La notion de profil est apparue dans le standard UML 1.3, comme un moyen qui permet de structurer les extensions UML (*tagged values, stereotypes et constraints*). Le mécanisme de profil inclus dans UML 2.0 définit un ensemble de concept de construction UML. Cela permet la spécification d'un modèle MOF pour traiter des concepts et des notations spécifiques qui sont nécessaires dans les domaines d'application particuliers. Les profils UML peuvent hériter d'autres profils, avoir des dépendances entre eux, ou encore être regroupés. L'OMG propose une liste importante des profils standardisés pour différents domaines.

Chacun de ces standards est en fait un profil UML spécifique à un domaine d'application ou à un environnement technique. Dans cette perspective, le profil UML est surtout perçu comme étant un mécanisme organisant les extensions UML, pour structurer UML en domaines d'applications spécifiques. Dans notre contexte de travail nous avons utilisé le profil *MARTE* [1], qui est un profil pour la modélisation et l'analyse des systèmes temps réel et embarqués [33].

2.4.2 Le profil MARTE

Le profil UML MARTE [33] est un standard de l'OMG. Ce profil est structuré autour de deux préoccupations, une pour modéliser les contraintes du temps réel et des systèmes embarqués et l'autre pour annoter des modèles d'application afin de supporter l'analyse des propriétés de système.

Le profil MARTE est constitué principalement de 3 paquetages favorisant la séparation des préoccupations, cf. figure 2.11 :

- le paquetage *foundation* : fournit les bases du langage qui traitent la modélisation des propriétés non-fonctionnelles, du temps, des ressources génériques et des allocations,
- le paquetage *design* permet de modéliser les applications et les plateformes d'exécution matérielles ou logicielles à différents niveaux d'abstraction,
- le paquetage *analysis* fournit les mécanismes permettant d'annoter les modèles à des fins d'analyse. Il fournit les concepts génériques pour couvrir tout type d'analyse et couvre nativement des analyses spécifiques comme les analyses d'ordonnançabilité ou de performance.

Un quatrième package *MARTE annexes*, contient les profils d'annexes définis dans MARTE, comme la bibliothèque des modèles prédéfinie *MARTE_ModelLibrary* (pour les types primitifs, les types de données prolongés, le temps, ...etc.) et le langage de spécification de valeur (VSL *Value Specification Language*). Le langage VSL fournit une syntaxe concrète claire permettant de saisir les propriétés non fonctionnelles des systèmes (poids, utilisation des ressources, consommation d'énergie, etc.) nécessaires aux analyses.

Dans ce qui suit, nous présentons les éléments de MARTE les plus importants pour la suite du document, i.e. les éléments utiles dans le contexte d'adaptabilité. Les concepts

1. *UML Profile for Modeling and Analysis of Real-time and Embedded Systems*

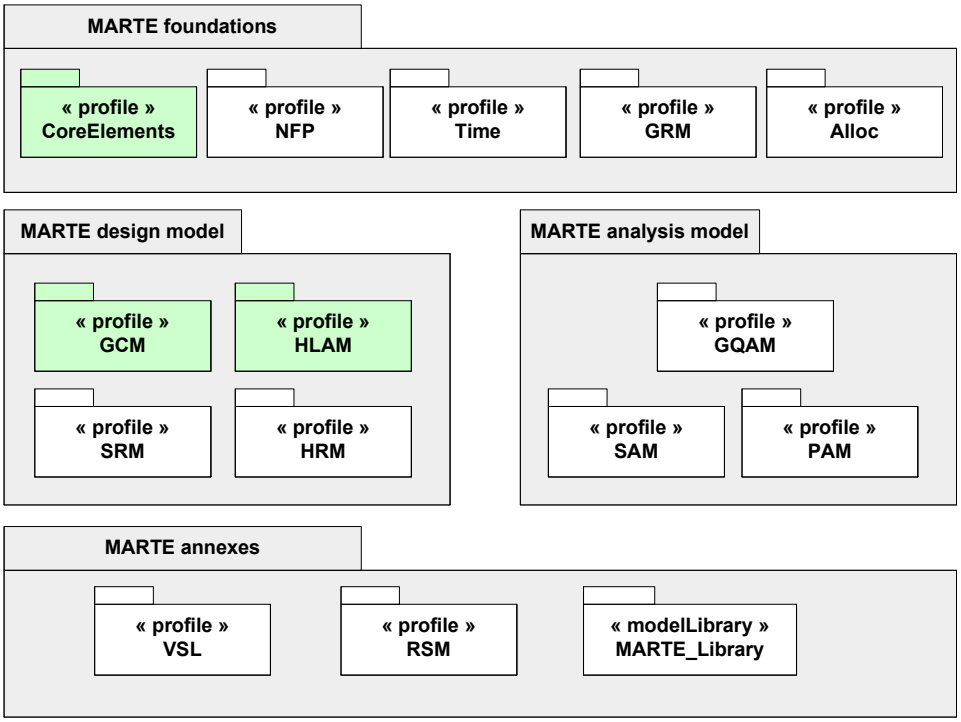


FIGURE 2.11 – L'architecture du profil UML MARTE

présentés dans le package *CoreElements* de *MARTE foundations* sont divisés en deux packages : *Foundations* et *Causality*. Le package *Causality* décrit les éléments de base nécessaires à la modélisation du comportement, et leur sémantique exécution.

La *CommonBehavior* de *Causality* définit le comportement modal, comme l'illustre la figure 2.12. Ce comportement modal est lié à la notion d'un mode opérationnel, ce qui peut représenter des choses différentes :

- un état d'un système opérationnel (ou sous système) qui est géré par des mécanismes de reconfiguration (par exemple, le middleware de gestion de tolérance aux fautes) selon les conditions de défaillance.
- un état d'un système opérationnel à un niveau donné de qualité de service qui peut être géré par l'infrastructure de gestion des ressources (par exemple, un middleware qui affecte les ressources à l'exécution en fonction de la charge de la demande, des contraintes temporelles, ou l'utilisation des ressources).
- une phase d'un système opérationnel, par exemple, démarrage, arrêt, le lancement, dans un système aéronautique.

Un mode identifie un segment opérationnel dans l'exécution du système qui se caractérise par une configuration donnée. La configuration du système peut être définie par un ensemble d'éléments du système actif (par exemple, les composants d'application, composants de la plateforme, les ressources matérielles), et / ou par un ensemble de paramètres de fonctionnement (par exemple, les paramètres qualité de service ou paramètres fonctionnels).

Un *BehavioredClassifier*, qui est un élément de méta modèle UML, peut être actif dans zéro ou plusieurs modes de fonctionnement, cf. figure 2.12. Par ailleurs, un *BehavioredClassifier* qui

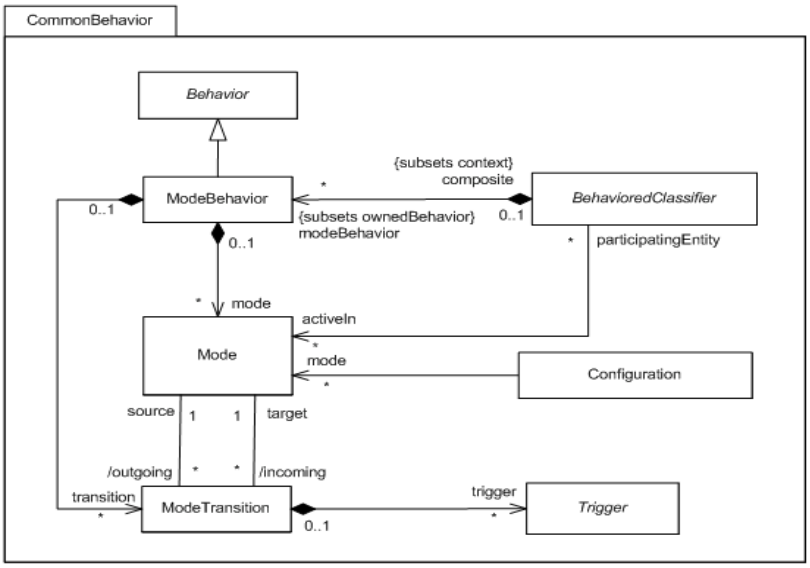


FIGURE 2.12 – Modèle du domaine de comportement modal de MARTE (source [33])

représente un système, sous-système ou toute entité composite peut avoir un ensemble de modes modélisé comme un *ModeBehavior*. Un *ModeBehavior* spécifie un ensemble de modes mutuellement exclusifs, c'est à dire, qu'un seul mode peut être actif dans un instant donné. En particulier, la dynamique des modes est représentée par l'interconnexion des modes par le biais de *ModeTransitions*.

Un *modeTransition* décrit le système modélisé sous un changement de mode. Un *modeTransition* se déclenche en réponse à un déclencheur, habituellement un événement indiquant le changement dans l'environnement. Dans la modélisation des propriétés non fonctionnelles (*NFPs*) de MARTE, une *NFP_Constraint* est définie comme une condition (une expression booléenne) sur les propriétés non-fonctionnelles associées aux éléments du modèle. Dans une architecture à base de composants, les composants peuvent supporter différents modes opérationnels, et ces modes opérationnels peuvent fournir différentes valeurs non-fonctionnelles ou qualités pour les mêmes services du composant. Ceci est représenté par l'association des *NFP_Constraint* au mode. Un *NFP_Constraint* donné peut également représenter le niveau de qualité dans plus d'un mode. Le niveau de qualité modélisé par un *NFP_Constraint* donnée dépend des ressources disponibles et les paramètres fonctionnels tels que les variables d'état qui permettent d'identifier la configuration du mode.

Le package *CoreElements*, cf. figure 2.13, définit comment les éléments du *Common-Behavior* étendent les métaclasses du métamodèle d'UML par les stéréotypes : *Mode*, *ModeTransition*, *ModeBehavior* et *Configuration*.

Dans le chapitre 4 et le chapitre 5 nous allons utiliser les trois packages GCM (*Generic Component Model*), HLAM (*High-Level Application Modeling*) et *CoreElements*. Le modèle de composant générique de MARTE (GCM) est principalement une extension du modèle de composant d'UML (abstraction des *structured classifier* d'UML). Ce modèle fournit un dénominateur commun entre différents modèles de composants, qui en principe ne ciblent pas exclusivement le domaine du temps-réel et de l'embarqué. Le but est de fournir dans

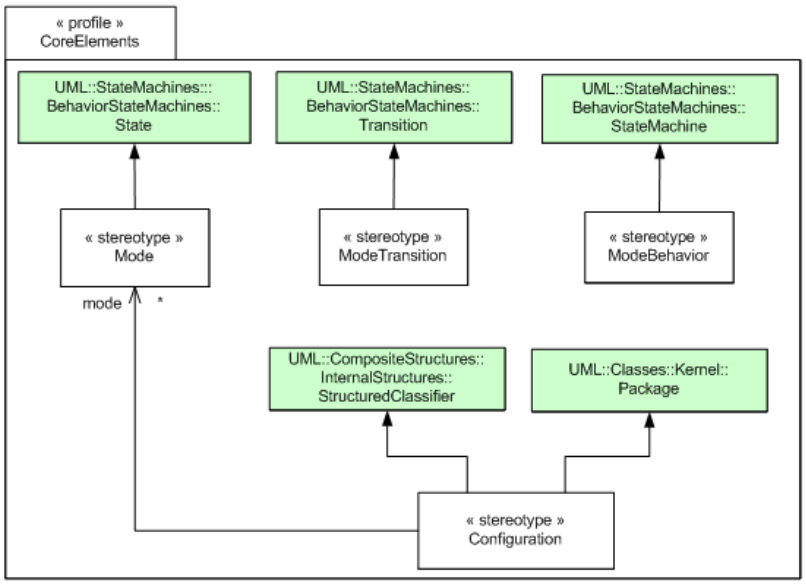


FIGURE 2.13 – Diagramme de profil UML pour la modélisation du *CoreElements* de MARTE (source [33])

MARTE un modèle aussi général que possible, qui n'est pas lié à une sémantique d'exécution spécifique, sur lequel des caractéristiques du temps réel peuvent être appliquées plus tard. Le modèle de composant générique de MARTE s'appuie principalement sur les *structured classifier* d'UML enrichi par un support pour les blocs SysML. La définition de certaines caractéristiques de GCM est influencée par le fait de fournir un support pour Lightweight-CCM, AADL et EAST-ADL2, cf. section 12.2.1 de la spécification de MARTE. Les types de port et de communication entre les composants GCM seront détaillés dans le chapitre 5.

L'objectif du package HLAM est de fournir des concepts de modélisation de haut niveau pour la modélisation des caractéristiques temps-réel et embarquées. En comparaison avec les domaines d'application habituels, le développement des systèmes temps réel nécessite des possibilités de modélisation d'une part des caractéristiques quantitatives telles que les délais et la période et, d'autres part, des caractéristiques qualitatives qui sont liées au comportement, communication et concurrence. Dans le chapitre 6 nous utilisons les stéréotypes *RtFeature* et *RtSpecification* pour la spécification des caractéristiques quantitatives (période).

2.5 Analyse des besoins

Nous analysons les besoins de notre travail selon les éléments nécessaires à la modélisation du comportement adaptatif et à l'analyse des contraintes de temps pour le comportement d'adaptation. Nous présentons également les besoins des systèmes adaptables en termes de modélisation d'architecture logicielle (modèle de structure, modèle de déploiement, les propriétés temporelles et les interfaces).

D'après la définition de l'adaptabilité (section 2.2.1) un système change suivant les changements de son environnement d'exécution. Cette définition nécessite la délimitation de

l'environnement, c.-à-d. l'identification et la modélisation des éléments de l'environnement pour lesquels le système est susceptible d'avoir un comportement d'adaptation. De même pour ce comportent d'adaptation, ses éléments de base doivent être spécifiés explicitement, notamment la variabilité et les règles d'adaptation (relation entre le changement de contexte et la sélection de la bonne variante). L'étape et la méthode de création de ce comportement représentent un point important en matière de coût de développement et de qualité de produit. Le comportent adaptatif peut être créé manuellement ou généré soit en phase de conception du système ou durant son exécution. L'adaptabilité entraîne des opérations (de reconfiguration) supplémentaires dans le système, ce qui nécessite la modélisation de ces opérations et leurs prises en compte lors de l'analyse temporelle du système. Ces éléments sont détaillés dans le tableau suivant table 2.1.

Élément de l'adaptabilité	**Significations**
Contexte	Modélisation des éléments de l'environnement pour lesquels le système est susceptible d'avoir un comportement d'adaptation
Variabilité	Modélisation des variantes des éléments du système, notamment : les implémentations, les connecteurs, le déploiement et les paramètres des composants ...etc.
Comportement adaptatif :	Modélisation de la relation entre le changement du contexte et la sélection de la bonne variante pour chaque composant
• modèle généré	Le modèle du comportement adaptatif est-il généré ?
• modélisé manuellement	Le modèle du comportement adaptatif est-il construit manuellement ?
• modélisé off line	Le modèle du comportement adaptatif est-il construit dans la phase de conception ?
• modélisé on line	Le modèle du comportement adaptatif est-il construit durant l'exécution du système ?
Opérations de reconfiguration	Modélisation des contraintes temporelles des opérations de reconfiguration fournies par le Framework pour l'adaptation
Contraintes temporelles du système	Modélisation des contraintes temporelles liées au fonctionnement des composants du système
Analyse temporelles du système pour son comportement adaptatif	L'analyse de l'ordonnançabilité du système avec la prise en compte de la charge temporelle des opérations de reconfiguration

TABLE 2.1 – Les éléments de base de l'adaptabilité

Le tableau précédent présente les éléments de base de l'adaptabilité. Ces éléments sont basés et liés au modèle d'architecture logicielle utilisé pour le système. L'architecture logicielle est définie par le modèle de structure, le modèle de déploiement, les interfaces et

les propriétés de temps. Afin de pouvoir prendre en compte les éléments de l'adaptabilité, l'architecture logicielle doit respecter quelques exigences présentées dans le tableau suivant 2.2 :

Élément de l'architecture logicielle	Exigences
Modèle de structure	Les éléments (composants) des systèmes doivent être définis explicitement. Afin de pouvoir définir leurs variantes et les replacer en exécution (la reconfiguration). Une définition explicite des interconnexions entre les éléments du système, avec un couplage faible entre ces éléments pour faciliter la reconfiguration, ce qui implique l'utilisation d'une approche à base de composant.
Déploiement	Le déploiement des composants ou de leurs instances doit être explicite pour pouvoir modéliser la variabilité de déploiement.
Propriétés temporelles	Les propriétés temporelles du système doivent être spécifiées explicitement afin de les prendre en compte lors de l'analyse du comportement adaptatif et pendant le déroulement normal.
Ports/Interfaces	De la même façon les interfaces doivent être spécifiées explicitement afin de pouvoir modéliser leur variabilité et pouvoir analyser le comportement adaptatif.

TABLE 2.2 – Les éléments de base pour l'adaptabilité

2.6 Conclusion

Dans ce chapitre, nous nous sommes intéressés à la présentation de la terminologie et les notions de base essentielles, qui facilitent la compréhension de la suite de ce mémoire.

Premièrement, nous avons présenté la définition de l'adaptabilité des systèmes, ses catégories et ses niveaux, ainsi que les types de changement possibles sur une configuration système. Ensuite, nous avons présenté des définitions qui sont liées aux systèmes temps réel et l'analyse d'ordonnançabilité. Dans le troisième point de ce chapitre, nous avons présenté l'ingénierie dirigée par les modèles, le langage de modélisation UML et son extension (le profil MARTE) pour la modélisation et l'analyse des systèmes temps réel embarqués. En dernier lieu, nous avons donné une analyse des besoins pour l'adaptabilité.

Selon ce que nous avons présenté dans ce chapitre, notre travail autour de la modélisation et l'analyse du comportement adaptatif des systèmes temps réel s'inscrit dans le cadre :

- Des systèmes temps réel embarqués à contraintes strictes.
- L'analyse d'ordonnançabilité, dans la phase de conception, des systèmes modélisés avec l'un des paradigmes : orienté objet ou flot de données synchrone.

- Dans notre travail nous optons pour la définition de l'adaptabilité qui considère que "*l'adaptation d'un système logiciel (S) est causée par le changement (ChE) d'un ancien environnement (E) à un nouvel environnement (E′) et aboutit à un nouveau système (S′) qui répond idéalement aux besoins de son nouvel environnement (E′)*"
- L'adaptabilité au niveau des applications (pas au niveau des systèmes d'exploitation ou niveau des middlewares).
- L'adaptabilité avec des plans de reconfiguration prédéfinis.
- Des changements sur le système de type *changement de structure*.
- L'utilisation du profil MARTE pour la modélisation et l'analyse.

CHAPITRE 3

L'état de l'art

3.1 Introduction

Ce chapitre présente un état de l'art des approches autour du développement des systèmes temps réel adaptables d'une manière générale et particulièrement pour le cadre de l'ingénierie dirigée par les modèles. Nous présenterons les approches existantes d'une manière dans laquelle nous nous concentrons davantage sur les éléments nécessaires à la modélisation du comportement adaptatif et à l'analyse des contraintes de temps pour ce dernier comportement. Ainsi, les approches seront présentées et comparées selon les éléments détaillés dans les tableaux 2.1 et 2.2. Nous concluions ce chapitre par un résumé dans lequel nous identifions les limites de chaque approche vis-à-vis des critères de classification et les objectifs cités dans l'introduction. De même, nous présentons nos choix pour les besoins liés à la modélisation des éléments de base de l'adaptabilité.

Dans la littérature il existe plusieurs approches et solutions pour le développement des systèmes temps réel adaptables. Ces approches peuvent être dans le cadre de l'ingénierie dirigée par les modèles ou non. Dans la suite, nous présentons principalement des travaux autour de développement des systèmes temps réel adaptables et nous focalisons sur les approches IDM dans lesquels s'inscrit notre travail. Parmi les solutions IDM existantes, nous trouvons des solutions destinées au développement des systèmes temps réel et d'autres pour le développement des systèmes adaptatifs, mais pas des solutions couplées pour les systèmes à la fois temps réel et adaptatifs. Dans ce qui suit nous présentons les approches de chaque catégorie. Nous présentons ensuite des travaux qui ne sont pas dans le cadre de l'ingénierie dirigée par les modèles, mais qui traitent le développement des systèmes temps réel et adaptables.

3.2 Adaptabilité et l'ingénierie dirigée par les modèles

Dans cette section, nous présentons les travaux les plus proches de notre contexte de travail et qui traitent principalement du développement des systèmes adaptables dans le cadre de l'ingénierie dirigée par les modèles. Ces travaux seront présentés selon les critères de comparaisons présentées dans le tableau 2.1.

3.2.1 MADAM

L'objectif global du projet MADAM (***m****obility and* ***ad****aptation-en****a****bling* ***m****iddleware*) [34] est de fournir aux ingénieurs logiciels des moyens adéquats pour développer des applications mobiles adaptatives.

Afin d'exprimer la variabilité, MADAM utilise les techniques de représentation et de dérivation utilisées dans les approches de famille de produit (*product family approaches* [35]). Les architectures de famille de produit décrivent les règles qui conduisent le calcul des variantes du système avec des propriétés différentes à partir d'un ensemble de composants communs. Ces règles prennent généralement la forme de points de variation dans l'architecture précisant les éléments optionnels ou alternatifs du système.

Dans MADAM, le modèle du système est une composition des composants types. Chaque composant type peut être atomique ou composite. Pour chaque composant type,

plusieurs implémentations peuvent être définies. Pour distinguer entre les alternatives d'implémentations d'un composant dans les modèles d'architecture de MADAM, ils annotent des composants avec des propriétés (*properties*). Les propriétés sont étroitement liées à des éléments de contexte. De cette façon, ils représentent les dépendances entre les implémentations de composants et leurs contextes. Ces dépendances sont représentées sur des propriétés indicatrices de fonctions (*property predictor functions*), ces propriétés attribuent des valeurs constantes aux propriétés du composant type afin de faire la liaison entre les implémentations du composant et les éléments du contexte d'adaptation, cf. figure 3.1. Le middleware automatise la dérivation d'une variante d'un contexte spécifique à l'exécution, c.-à-d. les configurations sont calculées en cours d'exécution.

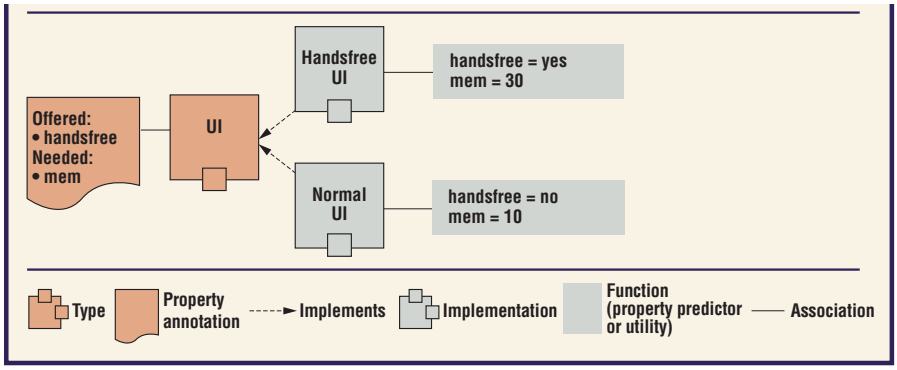


FIGURE 3.1 – La variabilité dans MADAM (source [34])

En résumé ce projet met l'accent sur tous les éléments de l'adaptabilité des systèmes, notamment : le contexte, la variabilité, le comportement adaptatif du système par rapport à son environnement. Cependant, la modélisation des opérations de reconfiguration, les caractéristiques temporelles du système et l'analyse temporelles du système pour son comportement adaptatif ne sont pas traitées dans le projet MADAM, cf. la tableau 3.1.

Légende : ✓ : supporté ≈ : supporté partiellement – : non supporté **Critères**	MADAM [34]
Contexte	✓ : le contexte dans MADAM est modélisé par des propriétés d'annotation (*Property annotation*).
Variabilité	✓ : afin d'exprimer la variabilité, MADAM utilise la notion de composant type et ses implémentations
Comportement adaptatif :	✓ : le comportement adaptatif est représenté par les dépendances entre les implémentations de composants et leurs contextes. Ces dépendances sont représentées sur des propriétés indicatrices de fonctions (*property predictor functions*), ces propriétés attribuent des valeurs constantes aux propriétés du composant type afin de faire la liaison entre les implémentations du composant et les éléments du contexte d'adaptation
• modèle généré	✓ : les configurations sont générées
• modélisé manuellement	– : les configurations ne sont pas crées manuellement.
• modélisé off line	– : les modèles de comportement ne sont pas créés en phase de conception.
• modélisé on line	✓ : les modèles de comportement sont créés en cours d'exécution
Opérations de reconfiguration	– : Dans MADAM les opérations de reconfiguration ne sont pas prises en compte dans les phases de modélisation.
Contraintes temporelles du système	– : MADAM ne permet pas de spécifier les caractéristiques temporelles pour les transition de la machine à états.
Analyse temporelles du système pour son comportement adaptatif	– : MADAM ne permet pas d'analyser les caractéristiques temporelles du système avec la prise en compte de l'influence de son comportement adaptatif

TABLE 3.1 – Présentation de MADAM selon les critères de la table 2.1

3.2.2 DiVA

Le projet DiVA (*Dynamic Variability in complex Adaptive systems*) [36, 37], utilise la modélisation orientée aspect (*AOM : aspect-oriented modeling*) et des modèles à l'exécution (*models at runtime*) pour la modélisation et la validation de l'adaptation dynamique. Le principal objectif est de modéliser des systèmes adaptatifs sans avoir à énumérer toutes leurs possibles configurations d'une manière statique, c'est-à-dire que les configurations ne sont pas énumérées à la phase de conception, mais en exécution. Pour atteindre cet objectif, une application est modélisée à l'aide du modèle de base (*base model*) qui contient les fonctionnalités communes et un ensemble de modèles de variantes (*variant models*) qui peuvent être composées avec le modèle de base, cf. figure 3.2. Le modèle de variantes capture la variabilité de l'application adaptative. Le modèle d'adaptation spécifie les variantes qui doivent être sélectionnées selon les règles d'adaptation et le contexte d'exécution du système. Ce modèle d'adaptation est construit à partir des exigences du système, raffiné lors de la conception et utilisé lors de l'exécution pour gérer l'adaptation. Donc dans ce projet, ils modélisent : le modèle de base, la variabilité, le contexte, les contraintes d'adaptation. Les configurations réelles de l'application sont construites à l'exécution par la sélection et la composition des variantes appropriées.

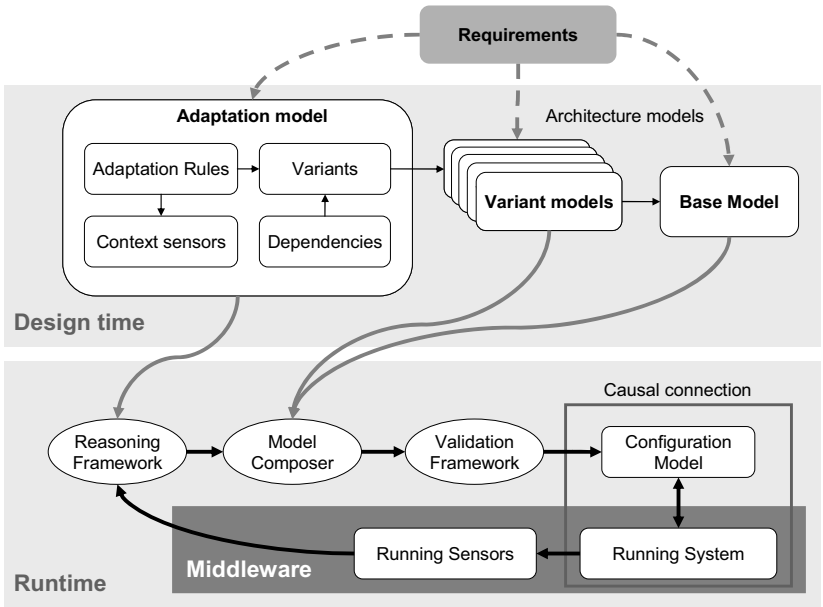


FIGURE 3.2 – L'approche DiVA pour la gestion de l'adaptation dynamique (source [36])

Dans la phase de conception, ils font une simulation pour construire un modèle d'adaptation en compagnie de toutes les possibles configurations. Cette simulation est basée sur un prototype et un simulateur implémenté en utilisant la plate-forme *Kermeta* [38]. Le principal avantage de la simulation est de permettre la validation des règles d'adaptation pendant la conception. En outre, le processus de simulation permet d'identifier les *live-locks* et *dead-locks* dans le graphe d'adaptation. *Dead-locks* dans le graphique de simulation correspondent à des cas où certaines règles d'adaptation conduisent à une configuration à partir de laquelle le système ne peut pas s'adapter. *Live-locks* correspondent aux cas où le système rebondit entre plusieurs configurations alors que le contexte ne change pas.

En résumé, le projet DiVA traite la notion de l'adaptabilité des systèmes dans une approche IDM en focalisant sur un traitement efficace du nombre de configurations possibles, qui peut croître de façon exponentielle avec chaque nouvelle dimension de variabilité. Dans cette approche on modélise : le contexte, la variabilité, le comportement adaptatif du système par rapport à son environnement. Cependant, la modélisation : des opérations de reconfiguration, les caractéristiques temporelles du système et l'analyse temporelles du système pour son comportement adaptatif, restent des points ouverts non traités dans le projet DiVA, cf. tableau 3.2.

Légende : ✓ : supporté ≈ : supporté partiellement – : non supporté **Critères**	DiVA [36]
Contexte	✓ : le contexte dans DiVA est modélisé par un ensemble de propriétés.
Variabilité	✓ : DiVA fait une séparation entre la partie qui ne change pas du système (*Base Model*) et la partie susceptible de changement (*Variant models*). Il définit toutes la variantes possibles pour la partie variables du système.
Comportement adaptatif :	✓ : le comportement adaptatif est représenté par des règles d'adaptation qui font la liaison entre la virante et son contexte d'utilisation.
• modèle généré	✓ : les configurations sont générées
• modélisé manuellement	– : les configurations ne sont pas crées manuellement.
• modélisé off line	– : les modèles de comportement ne sont pas créés en phase de conception.
• modélisé on line	✓ : les modèles de comportement sont créés en cours d'exécution
Opérations de reconfiguration	– : Dans DiVA les opérations de reconfiguration ne sont pas prises en compte dans les phases de modélisation.
Contraintes temporelles du système	– : DiVA ne permet pas de spécifier les caractéristiques temporelles pour les transition de la machine à états.
Analyse temporelles du système pour son comportement adaptatif	– : DiVA ne permet pas d'analyser les caractéristiques temporelles du système avec la prise en compte de l'influence de son comportement adaptatif

TABLE 3.2 – Présentation de DiVA selon les critères de la table 2.1

3.2.3 CEA-Frame

CEA-Frame [39] (*Construction and Execution of Adaptable applications*) une approche pour la construction et l'exécution des applications adaptables. CEA-Frame fournit principalement :

1. des méthodes pour la spécification des variantes d'une application en combinant les techniques de l'ingénierie dirigée par les modèles et la modélisation orientée aspect.
2. un mappage qui permet de générer les éléments du système liés à la plateforme (*platform specific level*) à partir de spécifications indépendant de la plateforme (*platform independent specifications*), cf. figure 3.3.

Dans le niveau *platform independent level*, un modèle primaire (*primary model*) et un ensemble de modèles d'aspect sont développées. Les variantes alternatives de l'application sont obtenues par l'utilisation des deux mécanismes suivants : i) la composition est utilisée pour calculer les variantes d'application en composant le modèle primaire avec les différents sous-ensembles des modèles d'aspect, et ii) les variantes de modèles d'aspect et de modèles primaires sont décrites au moyen de mécanismes de variabilité à base de modèle tels que la spécialisation et le paramétrage.

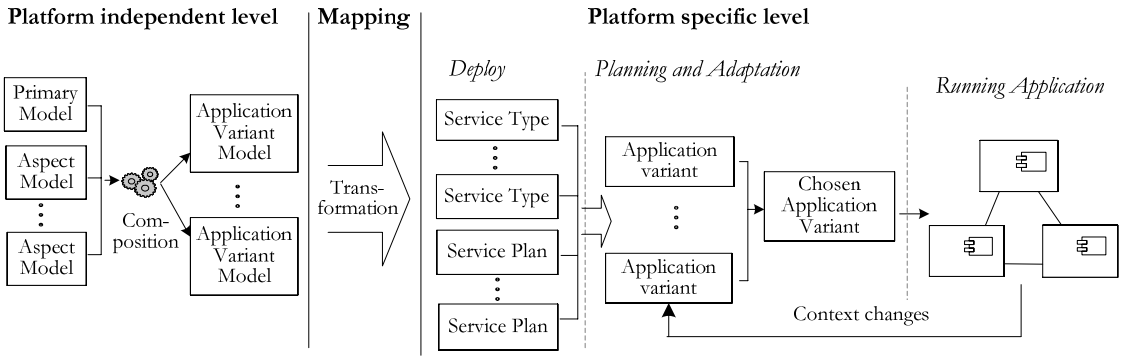


FIGURE 3.3 – Aperçu de l'approche CEA-Frame (source [39])

CEA-Frame présente également une partie pour l'instanciation et l'exécution de ses applications, mais nous nous limitons à la phase de modélisation et analyse de comportement adaptatif. En conclusion, cette approche modélise le contexte, la variabilité du système et son comportement adaptatif, mais ne modélise pas les opérations de reconfiguration et ne prends pas en compte l'influence de ces opérations sur les caractéristiques temporelles du système, cf. tableau 3.3.

Légende : ✓ : supporté ≈ : supporté partiellement – : non supporté **Critères**	CEA-Frame [39]
Contexte	✓ : le contexte dans CEA-Frame est modélisé par un ensemble de propriétés.
Variabilité	✓ : la variabilité est exprimée par la notion de spécialisation. Pour chaque élément type dans le système ses variantes sont définit comme des implémentations
Comportement adaptatif :	✓ : le comportement adaptatif est représenté par un modèle primaire qui représente la structure composite du système et les variantes possibles pour chaque éléments du système.
• modèle généré	✓ : les configurations sont générées
• modélisé manuellement	– : les configurations ne sont pas crées manuellement.
• modélisé off line	– : les modèles de comportement ne sont pas créés en phase de conception.
• modélisé on line	✓ : les modèles de comportement sont créés en cours d'exécution
Opérations de reconfiguration	– : Dans CEA-Frame les opérations de reconfiguration ne sont pas prises en compte dans les phases de modélisation.
Contraintes temporelles du système	✓ : CEA-Frame permet de spécifier les caractéristiques temporelles pour les transition de la machine à états.
Analyse temporelles du système pour son comportement adaptatif	– : CEA-Frame ne permet pas d'analyser les caractéristiques temporelles du système avec la prise en compte de l'influence de son comportement adaptatif

TABLE 3.3 – Présentation de CEA-Frame selon les critères de la table 2.1

Dans cette section nous avons présenté des travaux autour du développement des systèmes adaptables dans le cadre de l'ingénierie dirigée par les modèles. Ces travaux ne traitent pas le cas des systèmes temps réel dans le cadre de l'IDM. Dans ce qui suit nous présentons un panorama des travaux autour de l'adaptabilité pour les systèmes temps réel et dans le cadre de l'ingénierie dirigée par les modèles.

3.3 Temps réel et l'ingénierie dirigée par les modèles

Dans cette section, nous présentons les travaux les plus proches de notre travail et qui traitent principalement le développement des systèmes temps réel dans le cadre de l'ingénierie dirigée par les modèles afin de faire un choix pour la modélisation d'architecture logicielle qui répond aux exigences présentées précédemment dans le tableau 2.2. Ces travaux seront analysés selon les critères de comparaisons présentées dans le tableau 2.1.

3.3.1 EAST-ADL

EAST-ADL [40] est un langage de description d'architecture (*Architecture description language*) défini initialement dans le projet EAST-EEA et raffiné pour deux projets successifs ATESST1 et ATESST2 sous la forme d'un *Domain Specific Modeling Language* pour UML. EAST-ADL est un langage pour décrire des systèmes automobiles électroniques à travers un modèle d'information qui capture des informations d'ingénierie sous une forme standardisée. Les aspects abordés comprennent les caractéristiques des véhicules, les fonctions, les exigences, la variabilité, des composants logiciels des composants matériels et la communication.

Le modèle EAST-ADL est organisé selon quatre niveaux d'abstraction, cf. figure 3.4 :

1. ***Vehicle level*** : le niveau des véhicules représente le contenu et les propriétés du véhicule à partir d'une perspective de haut niveau sans pour autant exposer la réalisation.
2. ***Analysis level*** : il capture les principales interfaces et le comportement des sous-systèmes du véhicule. Il permet la validation et la vérification du système intégré ou de ses sous-systèmes à un haut niveau d'abstraction.
3. ***Design level*** : il comporte une définition fonctionnelle détaillée, une architecture matérielle et les allocations de fonctions pour le matériel.
4. ***Implementation level*** : la représentation de l'implémentation, n'est pas définie par EAST-ADL mais par AUTOSAR.

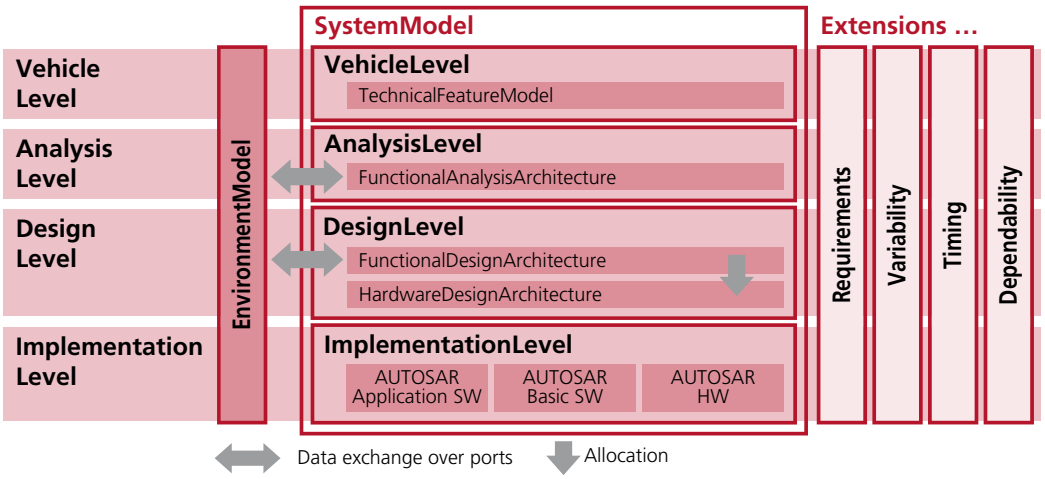


FIGURE 3.4 – Les niveaux d'abstraction et l'organisation du modèle d'EAST-ADL (source [40])

Afin de modéliser les éléments de chaque niveau d'abstraction le profil EAST-ADL définit le package *SystemModel*. Ainsi ce profil définit le package *FeatureModel* pour la modélisation des caractéristiques fonctionnelles, des contraintes ou des propriétés qui peuvent être présentes ou non dans une ligne de produits (de véhicule) pour tous les niveaux de modélisation. Le détail de la modélisation des caractéristiques qui sont spécifiques au niveau du véhicule (*vehicle level*) est pris en compte et documenté séparément dans le package *VehicleFeatureModeling*, plus de détails est fourni dans la spécification du profil [41].

Enfin, pour tous les niveaux d'abstraction, des éléments d'extension pertinents pour les exigences, le comportement, la variabilité (pour soutenir les familles de produits), le temps et la fiabilité sont associés à la structure de base.

L'extension du langage EAST-ADL pour le comportement contient la notion de *mode*, de déclencheur de fonction (*FunctionTrigger*), configuration et *FunctionBehavior*. Dans EAST-ADL le comportement de chaque fonction du système est spécifié séparément par *FunctionBehavior*. Ce qui est entendu par comportement est une fonction de transfert effectuant quelques calculs de données (en cas de *FlowPort interaction*) ou une opération qui peut être appelée par une autre fonction (dans le cas de l'interaction *ClientServer*). Un *FunctionBehavior* peut être attaché à plusieurs modes dans lesquels ce comportement peut être utilisé. Les *FunctionTrigger* sont des événements conditionnés pour l'activation d'une fonction dans un mode, ils n'activent pas les modes. Le *Behavior* (qui joue le rôle de Package en UML) regroupe les comportements des fonctions pour un contexte donné. Les modes d'EAST-ADL sont une manière d'introduire différentes configurations dans le système pour tenir compte des différents états du système, ou d'une entité matérielle ou une application. L'utilisation de modes leur permet de filtrer différentes vues du modèle. Les modes sont caractérisés par une condition booléenne fournie en tant que String, qui prend *true* comme valeur lorsque le mode est actif. La définition de mode dans MARTE est plus détaillée par rapport à celui d'EAST-ADL, en spécifiant les configurations de chaque mode. Le mode de MARTE est plus proche à la machine à état (étant une extension de celle-ci) ce qui lui permet de bien exprimer le comportement modal.

En ce qui concerne l'extension d'EAST-ADL pour la variabilité, elle comporte princi-

palement les notions : *ConfigurableContainer*, *ConfigurationDecision*, *ContainerConfiguration*, *VariableElement* et *VariationGroup*. Le *ConfigurableContainer* est une classe marqueur qui marque un élément identifié comme un élément configurable en mesure d'utiliser divers éléments optionnels (éléments variables). Le *ConfigurationDecision* représente une règle unique et atomisée sur la façon de configurer le modèle fonctionnel cible selon une configuration donnée du modèle fonctionnel source. Les *ConfigurationDecision* peuvent être regroupés par *ConfigurationDecisionFolder*. Le *ContainerConfiguration* définit une configuration réelle du contenu de la variable d'un *ConfigurableContainer*. Le *VariableElement* est une classe marqueur qui marque un élément comme étant optionnel, c'est-à-dire qu'il ne sera pas présent dans toutes les configurations du système complet. Le *VariationGroup* définit une relation entre un nombre arbitraire de *VariableElements*. Il est principalement destiné à définir la façon dont ces *VariableElements* peuvent être combinés (par exemple, l'un a besoin de l'autre, ou est alternatif de l'autre, etc.)

Selon le tableau de comparaison 3.4, EAST-ADL ne traite pas l'aspect d'adaptabilité. Notamment, la modélisation du comportement adaptatif et l'analyse temporelle du système pour ce comportement.

En ce qui concerne les exigences sur l'architecture logicielle pour l'adaptabilité, EAST-ADL offre tous les éléments nécessaires pour la modélisation de structure, de déploiement, les propriétés temporelles et l'utilisation des ports/interfaces.

Légende : ✓ : supporté ≈ : supporté partiellement – : non supporté **Critères**	EAST-ADL [40]
Contexte	✓ : le contexte dans EAST-ADL est modélisé par l'élément *Environment* qui est une collection des descriptions fonctionnelles de l'environnement. Le modèle *Environment* est utilisé pour décrire l'environnement de l'architecture électrique et électronique du véhicule. Il est modélisé par des fonctions continues représentant l'environnement du système.
Variabilité	✓ : EAST-ADL définit un package des éléments pour exprimer la variabilité dans le modèle du système. Ce package est basé principalement sur la notion de variabilité utilisée dans les familles de produits logiciels (*Software Product Families*)
Comportement adaptatif :	≈ : EAST-ADL utilise la notion de mode de fonctionnement mais ne traite pas le changement de mode explicitement. Il spécifie plusieurs variantes pour un élément. Pour chaque variante il indique son comportement qui est lié à un mode et le déclencheur de ce comportement.
• modèle généré	– : Le modèle de comportement n'est pas généré
• modélisé manuellement	– : Le modèle de comportement crée manuellement ne représente pas un modèle de comportement adaptatif vu que celui-ci ne concerne pas le comportement global du système envers son environnement, mais celui des fonctionnalités système envers l'environnement
• modélisé off line	– : Les modèles de comportement créés en phase de conception sont pour les fonctions et ne représentent pas le modèle de comportement adaptatif du système.
• modélisé on line	– : tous les modèles de comportement sont créés en phase de conception
Opérations de reconfiguration	– : Dans EAST-ADL les opérations de reconfiguration ne sont pas prises en compte dans les phases de modélisation.
Contraintes temporelles du système	✓ : EAST-ADL définit le package *Taming* qui permet de spécifier les caractéristiques temporelles pour chaque mode du système.
Analyse temporelles du système pour son comportement adaptatif	– : EAST-ADL ne permet pas d'analyser les caractéristiques temporelles du système avec la prise en compte de l'influence de son comportement adaptatif

TABLE 3.4 – Présentation d'EAST-ADL selon les critères de la table 2.1

3.3.2 Flex-eWare et FCM

Flex-eWare [42, 43] est un projet de recherche et de technologie financés par l'agence nationale française de recherche (ANR) et soutenu par les deux pôles de compétitivité régionaux : System@tic et Minalogic. Il vise à définir des outils open source pour le développement de logiciels dédiés à la flexibilité et des systèmes embarqués reconfigurables. Le projet consiste à :

- Améliorer deux familles de Framework de composants logiciels dédiés aux systèmes temps réel embarqués, à savoir Fractal et CCM et la mise en œuvre des capacités de reconfiguration des applications temps réel embarqués basées sur ces deux Framework.
- Fusionner le middleware de communication d'Ocarina et PolyORB-HI et adapter MyCCM pour cela.
- Combler l'écart entre Fractal et CCM en définissant un outil de modélisation commun et conforme à MARTE basé sur le modeleur UML2 Papyrus, de sorte que l'architecte d'applications sera en mesure d'assembler et de configurer ses composants logiciels Fractal ou CCM en utilisant soit le langage de modélisation dédié (voie 1 dans la figure 3.5, cette figure est focalisée sur les implémentations, ex. MyCCM à la place de CCM) ou en utilisant le langage de modélisation graphique commun (voie 2). Cet outil de modélisation est basé sur un méta-modèle commun, à savoir le modèle de composants Flex-eWare (FCM).
- Connexion de la chaîne d'outils OASIS à FCM.

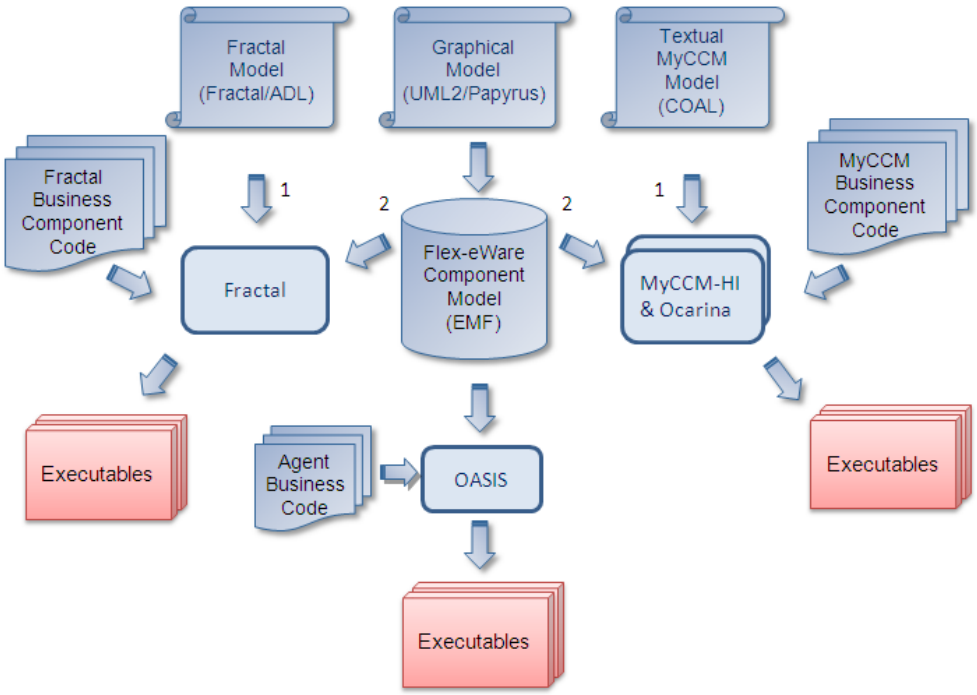


FIGURE 3.5 – Vue d'ensemble sur Flex-eWare (source [43]).

En résumé l'objectif du projet Flex-eWare est de définir un méta-modèle commun pour les applications à base de composants dans le domaine de l'embarqué et systèmes temps-réel. Ce méta-modèle FCM unifie le modèle de composant CORBA (CCM), Fractal et le

modèle de composant sous-jacent de l'environnement OASIS.

Ce qui nous intéresse dans ce projet c'est son modèle de composant FCM et le middleware eC3M de son implémentation. FCM comprend la structure composite et les communications entre les composants. La structure composite est constituée de la composition d'un composant à partir d'autres composants, en d'autres termes, un composant peut contenir d'autres composants. Chaque composant possède des interfaces fournies/requises. Les éléments de base dans FCM sont : *Component*, *Port*, *Connector*, *Part*, *Interface* et *PortKind*. Le composant (*Component*) représente une partie d'un système modulaire qui encapsule son contenu et dont la manifestation est remplaçable dans son environnement. Un composant définit son comportement en termes d'interfaces fournies et requises. À ce titre, un composant sert comme un type dont la conformité est définie par ces interfaces fournies et requises. Un composant peut être remplacé par un autre uniquement si les deux sont du type conforme. Ces interfaces fournies et requises forment la base pour les liaisons entre les composants en utilisant des connecteurs. Le connecteur (*Connector*) spécifie les liens de communication entre les composants. Le port (*Port*) est une propriété d'un classificateur et un point d'interaction entre ce classificateur et son environnement, ou entre le classificateur et ses parties (*Parts*). Une partie (*Parts*) est une propriété d'un classificateur et un élément structurel et réutilisable d'un modèle. Une interface (*Interface*) est un classificateur qui représente une déclaration d'un ensemble de caractéristiques publiques et cohérentes d'un composant. FCM définit le stéréotype *PortKind* qui représente un type de port. *PortKind* possède des règles qui sont mappées à un comportement bien défini.

FCM rempli toutes les exigences sur l'architecture logicielle pour l'adaptabilité,cf. 2.2, à savoir l'utilisation de la notion de composant basé sur un méta-modèle commun entre Fractal, CCM et MARTE, de même que pour son support pour la reconfiguration dynamique [44, 45]. Ce modèle de composant permet de modéliser explicitement les composants ainsi que leurs déploiements, propriétés temporelles, ports et interfaces, cf. tableau 3.5.

Élément de l'architecture logicielle	FCM de Flex-eWare [42]
Modèle de structure	Les composants FCM sont définis explicitement. Cela permet de définir leurs variantes et les replacer en exécution. Il offre également une définition explicite des interconnexions entre les éléments du système, avec un couplage faible entre ces éléments pour faciliter la reconfiguration.
Déploiement	Le déploiement des instances des composants FCM est explicite.
Propriétés temporelles	Les propriétés temporelles d'un système modélisé par FCM peuvent être spécifiées explicitement par le profil MARTE, vu que FCM support le modèle de composant GCM de MARTE.
Ports/Interfaces	Chaque composant FCM possède des interfaces fournies/requises et des ports spécifiés explicitement.

TABLE 3.5 – Les éléments de base pour l'adaptabilité dans FCM

3.3.3 CoSMIC

CoSMIC (*The Component Synthesis using Model Integrated Computing*) [46, 47] est un projet à l'universitaire de Vanderbilt dans l'institut ISIS (*Institute for Software Integrated Systems*), ce projet s'intéresse au développe des outils MDA, pour la composition et le déploiement des applications temps réel embarquées et distribuées à base de composant.

La modélisation de la structure dans CoSMIC est basée essentiellement sur le modèle de composant standard CCM (*CORBA Component Model*) [48]. CoSMIC s'appuie sur différents outils et formalismes pour le déploiement des applications à base de composants respectant les recommandations spécifiées dans le standard D&C (*Deployment & Configuration*) [49]. Les descriptions de l'assemblage des composants sont spécifiées à partir de la syntaxe graphique CADML (*Component Assembly & Deployment Modeling Language*), à partir duquel sont générés les plans de déploiement dans un format (XML) respectant le standard D&C. Pour configurer les applications réparties, CoSMIC utilise un langage spécifique : OCML (*Options Configuration Modeling Language*) qui est aussi un langage graphique. Il permet de définir les dépendances entre différentes options qui sont à sélectionner parmi une multitude d'options de configuration de l'application.

En résumé cette approche met l'accent seulement sur la configuration (composition) et le déploiement des composants. Cependant, la modélisation des points suivants n'est pas traitée dans la chaine d'outils CoSMIC : contexte, de la variabilité, du comportement adaptatif de système par rapport à son environnement, des opérations de reconfiguration et l'analyse temporelles du système pour son comportement adaptatif.

Nous avons présenté CoSMIC pour voir ce qu'il offre pour la modélisation des éléments de base pour l'adaptabilité. CoSMIC utilise le modèle de composant CORBA CCM, ce qui permet de modéliser la structure du système dans une approche à base de composant dont

les composants, les interfaces et le déploiement sont modélisés explicitement, cf. tableau 3.6. En ce qui concerne les propriétés temporelles, CoSMIC utilise *real-time CORBA*.

Élément de l'architecture logicielle	**CoSMIC [46, 47]**
Modèle de structure	La modélisation de la structure dans CoSMIC est basée essentiellement sur le modèle de composant standard CCM (*CORBA Component Model*).
Déploiement	Pour le déploiement CoSMIC respecte les recommandations spécifiées dans le standard D&C (*Deployment & Configuration*)
Propriétés temporelles	CoSMIC utilise *real-time CORBA* pour spécifier les propriétés temporelles, .
Ports/Interfaces	L'utilisation de modèle de composant CCM de CORBA permet à CoSMIC de spécifier les ports et les interfaces explicitement.

TABLE 3.6 – Les éléments de base pour l'adaptabilité dans FCM

3.3.4 La suite d'outils Fujaba

La suite d'outils Fujaba[1] est un outil CASE[2] libre, offrant aux développeurs un support pour l'ingénierie et la réingénierie logicielle à base des modèles. Le projet a démarré au sein de groupe de l'ingénierie logicielle de l'université de Paderborn en 1997 dénommé Fujaba, et actuellement, il est développé par plusieurs universités réparties principalement en Allemagne et quelques autres pays. Il a été conçu en tant qu'une application monolithique comprenant plusieurs fonctionnalités de différents domaines. En 2002 Fujaba a été reconçu et est devenu la suite d'outils Fujaba (Fujaba tool suite). Il fournit maintenant une architecture modulable (plug-in), qui permet aux développeurs d'ajouter facilement des fonctionnalités tout en conservant un contrôle total sur leurs contributions. Depuis 2006, il existe une intégration de Fujaba dans la plateforme Eclipse. Différents modules d'extension (plug-in) existent, dont : un module pour **la modélisation, la validation et la vérification des systèmes temps réel embarqués** [50, 51]. Ce module fournit un langage de modélisation spécifique (MECHATRONIC UML) pour la capture des modèles [52]. Ce dernier est une extension d'UML pour la modélisation et l'analyse des systèmes mécatroniques (combinaison de systèmes mécaniques, électroniques et logiciels temps réel). En effet, la partie structurelle du système est modélisée par les diagrammes de composant et de classe. Le papier [53] présente l'approche utilisée dans Fujaba pour la spécification de l'architecture et de la communication temps réel complexe entre les composants par des diagrammes de composants UML et les patrons (patterns) respectivement. Concernant la modélisation du comportement du système, des machines à états étendues (*Real-Time Statecharts* [54]) sont utilisées. Leur extension permet principalement de modéliser le

1. Fujaba : ***F**rom **U**ML to **J**ava **a**nd **b**ack **a**gain*
2. CASE tools : *Computer-aided software engineering tools*

temps consommé/nécessaire pour l'exécution d'une transition de la machine à états. Cette approche permet de fixer le temps maximal d'exécution (WCET) de transition. Les différents modes possibles pour le système sont définis par les états. Le déclenchement d'une transition n'engendre pas forcément le passage de son état source à son état cible. Il est possible de définir un état intermédiaire abstrait (*Answer*) à partir duquel le système peut accepter la transition et passer à l'état cible ou refuser la transition et rester dans l'état source.

Quant à l'aspect de reconfiguration, en plus au comportement de chaque composant, des règles de coordination entre ces composants sont définies. Durant l'exécution, les composants changent de comportement selon leurs états et les règles de coordinations. Par ailleurs, la modélisation des points suivants est restée ouverte et non traitée dans la chaine d'outils Fujaba, cf. tableau 3.7 : contexte, de la variabilité, du comportement adaptatif du système par rapport à son environnement, des opérations de reconfiguration et l'analyse temporelles du système pour son comportement adaptatif.

Légende : ✓ : supporté ≈ : supporté partiellement – : non supporté **Critères**	Fujaba suite tool [50]
Contexte	– : le contexte dans Fujaba n'est pas prises en compte dans les phases de modélisation
Variabilité	– : Dans Fujaba les variantes possibles pour le système ne sont pas prises en compte dans les phases de modélisation
Comportement adaptatif :	≈ : Fujaba utilise la notion de mode de fonctionnement, mais ne traite pas le changement de mode selon la définition de l'adaptabilité. La reconfiguration est faite par une demande (envoie de message) qui peut être acceptée ou non et non pas par l'observation de changement de l'environnement.
• modèle généré	– : le modèle de comportement n'est pas généré
• modélisé manuellement	✓ : le modèle de comportement est crée manuellement .
• modélisé off line	✓ : les modèles de comportement sont créés en phase de conception.
• modélisé on line	– : les modèles de comportement sont créés en phase de conception
Opérations de reconfiguration	– : Dans Fujaba les opérations de reconfiguration ne sont pas prises en compte dans les phases de modélisation.
Contraintes temporelles du système	✓ : Fujaba permet de spécifier les caractéristiques temporelles pour les transition de la machine à états.
Analyse temporelles du système pour son comportement adaptatif	– : Fujaba ne permet pas d'analyser les caractéristiques temporelles du système avec la prise en compte de l'influence de son comportement adaptatif

TABLE 3.7 – Présentation de Fujaba selon les critères de la table 2.1

3.4 Adaptabilité et le temps réel

Dans la littérature, plusieurs travaux abordent le sujet de changement de mode pour les systèmes temps réel, cependant ils ne traitent pas la notion de l'adaptabilité telle qu'elle est utilisée dans notre travail. Toutefois, le changement de mode étant au cœur de notre problématique il est intéressant de fournir un aperçu de ces travaux. L'étude de Jorge Real [55] présente un état de l'art sur les protocoles de changement de mode pour les systèmes temps réel. Nous retiendrons quatre travaux représentatifs :

- Pedro et Burns fournissent dans leur travail [56] un nouveau modèle pour les changements de mode dans les systèmes temps réel flexibles et l'analyse d'ordonnançabilité liée à ces changements de mode. Ils ont introduit un protocole de changement de mode dans l'architecture du système afin de passer d'un mode à un autre d'une manière contrôlée. Entre les modes, ils changent les fonctionnalités du système et garantissent l'ordonnançabilité des tâches qui débutent dans l'ancien mode et continuent l'exécution pendant la durée de changement de mode, pareillement pour les tâches qui sont introduites lors du changement de mode. En d'autres termes, ils analysent l'ordonnançabilité des tâches au cours d'un changement de mode, sans avoir pris en compte les opérations de reconfiguration dynamique. Ces dernières ne sont pas nécessaires dans leur contexte de travail parce qu'ils utilisent les variables partagées pour la communication entre les tâches. Ce travail ne parle pas de la modélisation du contexte, les opérations de reconfiguration, le comportement adaptatif et son analyse.
- Dans le travail de thèse d'Etienne Borde [57], chacun des comportements possibles du système est représenté par un mode de fonctionnement auquel est associée une configuration logicielle. Ces modes représentent les nœuds d'un automate de comportement. Il utilise la spécification des règles de transition entre ces modes de fonctionnement afin de générer l'implantation des mécanismes de changement de mode. Le code ainsi produit respecte les contraintes de réalisation des systèmes critiques et implante des mécanismes de reconfiguration sûrs et analysables. Dans ce travail, les modes et les configurations associées sont modélisés manuellement et ne font aucune liaison avec les éléments de l'environnement d'exécution (contexte). La modélisation du contexte, des opérations de reconfigurations et de la variabilité dans le système sont des points non traités dans ce travail. Quant à l'analyse logicielle liée à la reconfiguration, l'approche calcule le temps maximal requis pour passer d'un mode de fonctionnement donné à un autre mode de fonctionnement afin de le prendre en compte dans une analyse d'ordonnançabilité.
- TimeAdapt [58, 59] est une approche pour supporter la reconfiguration dynamique limitée par le temps (*timely reconfigurations*) pour des applications temps réel embarquées à base de composants. L'approche est divisée en trois parties : le langage de spécification de reconfiguration, l'estimateur de temps d'exécution de reconfiguration et l'exécuteur de reconfiguration. Les reconfigurations possibles sont spécifiées en utilisant un langage de reconfiguration et elles servent comme entrée pour le test d'admission (l'estimateur). Le test d'admission renvoie une probabilité liée au fait qu'une reconfiguration donnée peut satisfaire son échéance spécifiée ou non. Si la probabilité dépasse un seuil donné, la reconfiguration est ordonnancée comme une tâche temps réel de haute priorité et ses actions de reconfiguration sont exécutées.

Dans le cas contraire, la reconfiguration est réordonnancée avec une nouvelle échéance à une date ultérieure. Cette technique est destinée aux systèmes à contrainte de temps souple en offrant un moyen pour exécuter les opérations de reconfigurations dans un temps limité, cependant elle ne modélise pas le comportement adaptatif et ne prends pas compte son influence sur les caractéristiques temporelles du système.

– Zhang propose dans [60] une approche de vérification de modèle modulaire, afin de vérifier que le modèle formel d'un programme adaptable, spécifié par une logique temporelle linéaire (*LTL : Linear Temporal Logic*), satisfait ses exigences spécifiées dans A-LTL (une extension pour l'adaptation de LTL). Dans cet article, les auteurs séparent les préoccupations fonctionnelles des préoccupations d'adaptation. Plus précisément, ils modélisent un programme adaptatif comme une collection de programmes dans un état stable (*steady-state programs*) et une série d'adaptations qui réalisent les transitions entre les programmes stables en réponse aux changements de l'environnement. Ils utilisent LTL pour spécifier les propriétés des parties non adaptatives du système et ils utilisent ALTL pour spécifier les propriétés (*transitional properties*) qui conservent durant le processus d'adaptation. Dans cette approche les auteurs vérifient que chaque *steady-state program* (une configuration dans notre cas) respecte ses propriétés locales et des conditions sur le comportement fonctionnel du système. Cependant, ils ne valident pas les contraintes temps réel pendant les opérations de reconfiguration.

3.5 Synthèse et conclusion

Dans ce chapitre de l'état de l'art, nous avons présenté l'ensemble des travaux de littérature liés au thème de notre thèse : le développement des systèmes temps réel adaptable. Cette présentation a été basée sur des critères de comparaison et classification de ces travaux définis précédemment dans le tableau 2.1. Le tableau 3.8 propose à titre de synthèse une classification des travaux de littérature présentés en fonction de ces critères de comparaison.

La majorité des travaux existants [56, 57, 55, 60] sont focalisés sur les protocoles de changement de mode sans prises en compte des opérations de reconfiguration. Ces travaux ne traitent pas le problème de la modélisation du comportement adaptatif et n'utilisent pas l'ingénierie dirigée par les modèles. Cependant, notre travail est complémentaire à ces travaux de changement de mode. Les protocoles de changement de modes s'inscrivent dans le cadre des techniques de reconfiguration dynamique pour les systèmes temps réel, c'est-à-dire des techniques de mise en œuvre de la reconfiguration. Alors que notre travail est focalisé sur l'étape de la modélisation et de la validation du comportement adaptatif et non pas à la mise en œuvre des techniques de reconfiguration. Tel que présenter dans la définition de l'adaptabilité (section 2.2.1), l'adaptabilité est composée de trois fonctions : gestionnaire du contexte (*EvenChangeRecognition*), gestionnaire du comportement d'adaptation (*SysChangeRecognition*) et le gestionnaire de reconfiguration (*SysChange*). Notre travail est complémentaire aux travaux de changement de mode de façon que ces derniers peuvent être utilisés lors de la mise en œuvre du gestionnaire de reconfiguration. Les travaux [40, 62, 63, 46, 50, 61] qui traitent le développement des systèmes temps réel dans le cadre de l'IDM ne traitent pas les problèmes liés à la modélisation du comportement adaptatif et le changement de mode. Enfin, les solutions [34, 36, 37, 39] existantes pour la

Légende : ✓ : supporté ≈ : supporté partiellemer – : non supporté **Critères**	**Approches et outils**	Fujaba suite tool [50]	CoSMIC [46]	EAST-ADL [40]	MADAM [34]	DiVA [36]	CEA-Frame [39]	Pedro et Burns [56]	Étienne Borde [57]
Contexte		–	–	✓	✓	✓	✓	–	–
Variabilité		–	–	✓	✓	✓	✓	–	–
Comportement adaptatif :		≈	–	–	✓	✓	✓	✓	✓
• modèle généré		–	–	–	✓	✓	✓	–	–
• modélisé manuellement		✓	–	–	–	–	–	✓	✓
• modélisé off line		✓	–	–	–	–	–	✓	✓
• modélisé on line		–	–	–	✓	✓	✓	–	–
Opérations de reconfiguration		–	–	–	–	–	–	–	–
Contraintes temporelles du système		✓	✓	✓	–	–	✓	✓	✓
Analyse temporelles du système pour son comportement adaptatif		–	–	–	–	–	–	✓	≈

TABLE 3.8 – Classification des approches selon les critères de la table 2.1

gestion du comportement adaptatif dans une approche IDM ne prennent pas en compte les contraintes de temps du système et ne font pas la validation de ces contraintes vis-à-vis du comportement adaptatif. Ces méthodes ne traitent pas la modélisation et la validation du comportement adaptatif en phase de conception, mais ils génèrent les configurations en cours d'exécution. À notre connaissance il n'existe pas de solution qui offre les moyens pour modéliser et gérer le comportement adaptatif en phase de conception et valider ce comportement vis-à-vis des contraintes temporelles de son système. Vu que l'aspect de l'adaptabilité pour les systèmes temps réel n'est pas traité dans le cadre de l'IDM, cela a nécessité de faire un état de l'art plus large et de présenter des travaux qui ne traitent pas directement ce problème.

Aucune des approches de la littérature ne satisfait tous ces critères de comparaison. Une façon de valoriser notre thèse est de définir une approche de modélisation et validation qui satisferait tous les points listés ci-dessus : relatifs à la modélisation et la gestion du comportement adaptatif et à la validation de ce comportement vis-à-vis des contraintes temporelles du système. Dans le chapitre suivant nous proposons une approche pour la modélisation et la gestion du comportement adaptatif.

En ce qui concerne le modèle d'architecture utilisé pour le système, nous avons opté pour un modèle de composant. Le modèle de composant choisi est le modèle FCM développé dans le laboratoire (LISE [64]) dans le cadre du projet collaboratif Flex-eWare [43]. Ce choix est basé sur les avantages offerts par ce modèle de composant. FCM rempli toutes les exigences sur l'architecture logicielle pour l'adaptabilité présentées dans le tableau 2.2. Ce modèle de composant permet de modéliser explicitement les composants ainsi que leurs déploiements, les propriétés temporelles, les ports et interfaces. Il est présenté ci-après en

introduction du chapitre 4.

CHAPITRE 4

Le comportement adaptatif

4.1 Introduction

Dans les parties précédentes nous avons présenté le domaine de la modélisation et de l'analyse du comportement adaptatif des systèmes temps réel embarqués dans le cadre de l'ingénierie dirigée par les modèles, ses problématiques ainsi qu'un état de l'art des travaux liés à ce domaine. La modélisation et l'analyse du comportement adaptatif constitue l'une des contributions de cette thèse. Dans ce chapitre, nous proposons tout d'abord une approche de modélisation du comportement adaptatif basée sur le profil UML MARTE. Puis nous proposons deux manières d'automatiser la création du modèle de comportement adaptatif. L'objectif de cette approche est de faciliter la création et la gestion du modèle de comportement adaptatif afin de pouvoir l'analyser. Nous présentons les principes sur lesquels s'appuie notre approche et donnons une vue globale de son fonctionnement avant de détailler successivement ses différentes composantes.

La première question à traiter pour le développement d'un système adaptatif basé sur des plans d'adaptation prédéfinis est la modélisation du comportement d'adaptation et la gestion de toutes les configurations qui peuvent être utilisés durant l'exécution du système. Dans ce qui suit, nous présentons la modélisation du comportement adaptatif en utilisant *Modal Behavior* de MARTE et le modèle de composant FCM. Ces formalismes de modélisation sont supportés par un environnement de développement constitué d'un modeleur UML, Papyrus, et d'un outil de déploiement des applications, eC3M [1] [65].

4.2 La modélisation du comportement adaptatif

La modélisation du comportement adaptatif dans notre approche est fondée sur la notion de Mode du profil UML *MARTE* (voir section 2.4.2 pour plus d'informations sur *Mode* et *MARTE*).

L'exemple de la section 7.3.3 de la spécification de MARTE [33] présente un système reconfigurable qui utilise les concepts *Mode* (opérationnel) et *configuration*, comme le montre les deux figures suivantes 4.1 et 4.2 par les stéréotypes appliqués sur la structure composite et les nœuds de la machine à état. La figure 4.1 présente un exemple de spécification du comportement modal en utilisant une machine à état stéréotypée par *modeBehavior* de MARTE. Cet exemple modélise une application logicielle qui possède deux modes de fonctionnement *NominalMode* et *DegradedMode*, représentés par deux états stéréotypés par *Mode* de MARTE.

1. embedded Component Container Connector Middleware

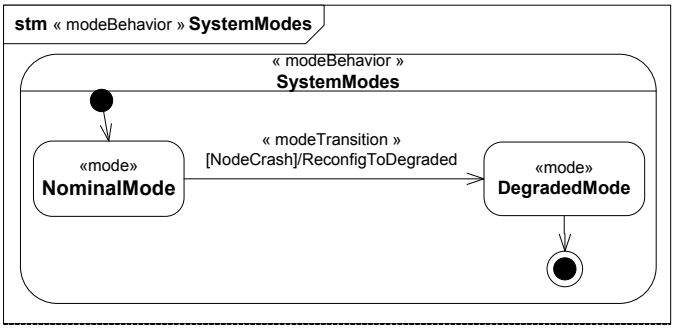


FIGURE 4.1 – modélisation de mode en MARTE (source [33])

Les informations nécessaires pour le changement de modes, tels que les événements déclencheurs et les effets de passage entre modes sont modélisées par l'élément UML Transition et stéréotypés par *modeTransition* de MARTE. La configuration du système pour le mode *DegradedMode* est représentée en utilisant une structure composite avec l'ensemble des composants et leurs interconnexions, cf. figure 4.2. L'exemple présente également un scénario d'allocation des composants de la configuration système (élément de la structure composite) sur un ensemble des ressources de la plate-forme logicielle/matérielle. Afin d'indiquer que cette configuration est valide pour le mode *DegradedMode* l'attribut *Mode* du stéréotype *configuration* est utilisé.

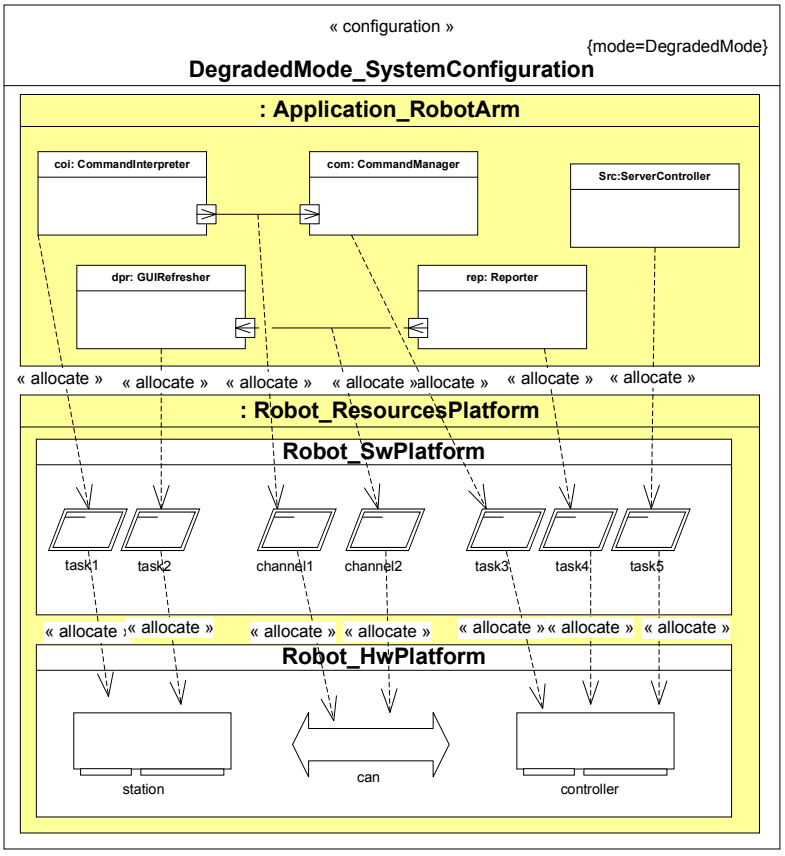


FIGURE 4.2 – modélisation de configurations en MARTE (source [33])

En résumé, le modèle du comportement modal dans MARTE est constitué principalement d'une machine à état dont chaque état représente un mode. À chaque mode est attachée une configuration du système qui est défini pour un environnement bien déterminé. Les transitions représentent les passages possibles entre modes. Cependant, MARTE ne propose pas une méthodologie de modélisation du comportement adaptatif et ne fournit pas les moyens de modélisation des éléments de l'environnement pour lesquels le système peut exhiber un comportement d'adaptation. Autrement dit, MARTE ne présente pas comment les informations de *ModalBehavior* font la liaison avec les éléments de l'environnement ? Également, il ne présente pas aussi comment les événements déclencheurs et les effets des (passages entre configurations) transitions sont-ils identifiés ? Sans compter que l'utilisation du *ModalBehavoir* pour la création et la modification du modèle du comportement adaptatif est difficile en raison du grand nombre des configurations possibles pour les systèmes. Afin de surmonter ces difficultés, nous proposons deux solutions pour l'automatisation de la création du comportement adaptatif en utilisant le *ModalBehavior* et *NFP_Constraint* de MARTE et d'autre stéréotype pour la variabilité. Dans ce qui suit, nous présentons les éléments de base pour la modélisation du comportement adaptatif.

Tel que montré précédemment (dans la section 2.2.1), un système est adaptable s'il dispose d'une fonction d'adaptation, où l'adaptation signifie le changement du système afin d'accommoder le changement de son environnement. Par conséquent, les informations contenues dans le modèle de développement du système adaptatif doivent décrire les informations nécessaires pour le bon fonctionnement de son comportement adaptatif. Cela concerne notamment, le contexte, les mécanismes d'adaptation fournis par le middleware, la variabilité des implémentations des composants et les règles d'adaptation.

– **Contexte** : le modèle du contexte est la représentation minimale de l'environnement auquel le système doit être adapté. Le contexte est modélisé par une ou plusieurs classes UML dont chaque attribut d'une classe capture un élément de l'environnement. La figure 4.3 présente un exemple de classe du contexte. La capacité de la batterie est représentée par une variable limitée entre 0 et 100 pour indiquer la capacité en pourcentage. L'état de la disponibilité d'une liaison wifi est représenté par une variable booléenne.

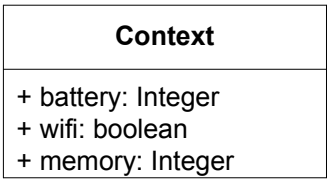


FIGURE 4.3 – Représentation de l'environnement par la classe *Context*

– **Middleware de reconfiguration** : dans ce modèle nous modélisons les services et opérations fournis par le middleware pour la reconfiguration dynamique. Tous les services (ex. *change*, *remove*, ... etc.) sont modélisés sur une ou plusieurs classes UML sous forme des opérations et chaque opération est annotée avec MARTE pour spécifier ses caractéristiques temporelles, notamment le pire temps d'exécution (WCET : *worst case execution time*). La figure 4.4 illustre un exemple de framework de reconfiguration composé de 3 opérations *Change*, *Add* et *Remove*. Le *wcet* de *Add* est de 3 ms.

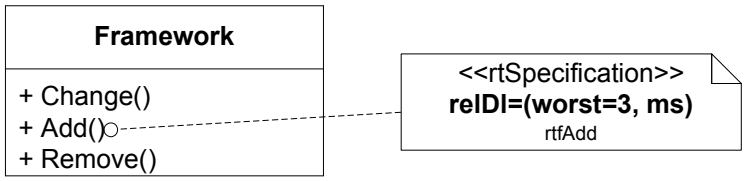


FIGURE 4.4 – Représentation du framework de reconfiguration par la classe *Framework*

– **La variabilité des implémentations des composants** : ce modèle représente les alternatives et les variantes possibles des éléments du système. Autrement dit, chaque composant dans le système susceptible d'utiliser plus d'une implémentation en exécution représente un point de variation (élément variable) dans le système. Chaque implémentation représente une variante. Pour chaque variante sont attachées des conditions qui indiquent quand est ce qu'elle doit être utilisée en fonction du contexte. Par exemple, la figure 4.5 présente deux composants C_1 et C_2 et leurs variantes C_1^1, C_1^2 et C_2^1, C_2^2 respectivement.

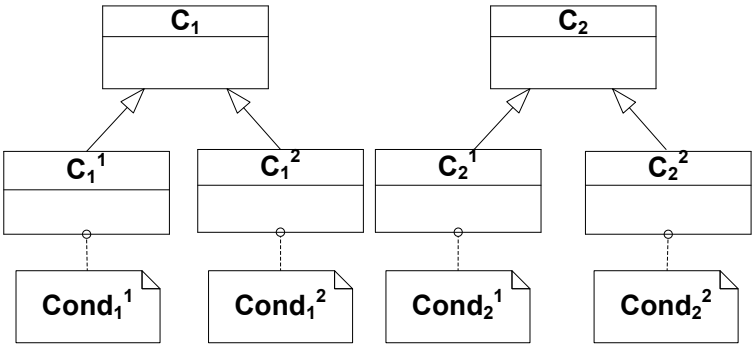


FIGURE 4.5 – Représentation de la variabilité de deux composants

– **Les règles d'adaptation** : les règles d'adaptation spécifient comment le système devrait s'adapter à son environnement. Une règle est représentée par une condition d'utilisation d'une variante. Les conditions font relation entre les variantes et les variables de l'environnement. Par exemple, la figure 4.6 présente une condition d'utilisation de la variante C_1^1 du composant $C1$, cette condition exige l'utilisation de cette variante lorsque la capacité de la batterie dépasse 50 %. Les conditions peuvent être complémentaires, redondantes ou en opposition. Les conditions des variantes d'un élément variables doivent couvrir la plage des valeurs possibles pour la variable d'environnement utilisée par ses conditions, plus de détails cf. 4.4.2.

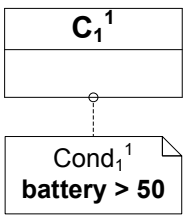


FIGURE 4.6 – Une règle d'adaptation

Le méta-modèle donné figure 4.7 présente le concept de mode de MARTE. Il permet d'identifier un cadre d'usage du concept. Le comportement modal est identifié par le concept de *ModeBehavior*. C'est une extension de la machine à état UML. Pour chaque état de

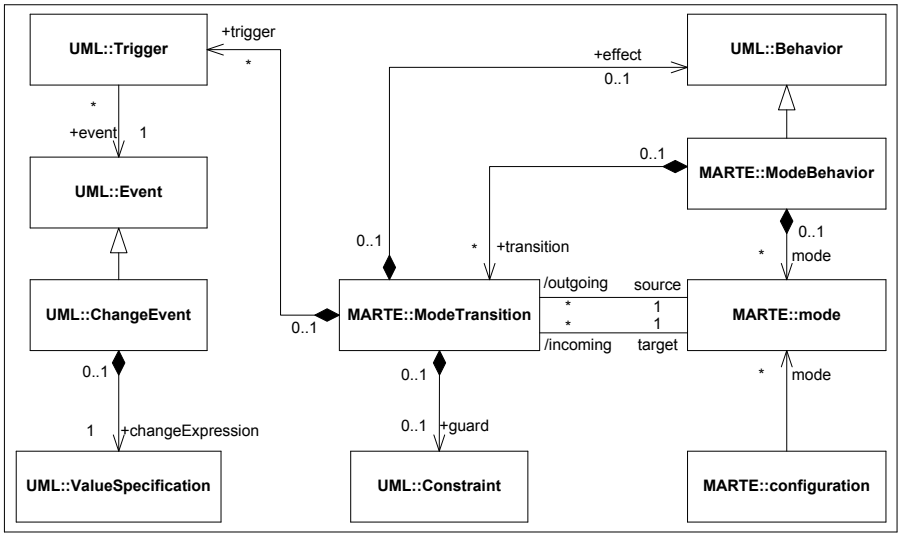


FIGURE 4.7 – Utilisation de *Modal Behavior* de MARTE pour exprimer le comportement adaptatif

l'environnement, une configuration est définie et associée à un mode. Une configuration est définie par le choix d'une variante pour chaque élément variable dans le système. Chaque fois qu'une variante est sélectionnée pour la configuration, ses conditions d'utilisation sont ajoutées aux conditions de délimitation de l'état de l'environnement dans lequel cette configuration sera utilisée. Les modes sont reliés par *ModeTransition*. Pour les deux bouts de chaque transition deux modes sont utilisés l'un comme source et l'autre comme cible. La transition peut avoir plusieurs déclencheurs (*Trigger*). Chaque déclencheur est activé suite à l'occurrence d'un événement qui est lié au résultat booléen de l'évaluation d'une expression UML. Par exemple, la figure 4.8 présente le comportement adaptatif d'un système sous forme de 4 modes. Pour chaque mode nous définissons une configuration composée d'une combine spécifique des variantes.

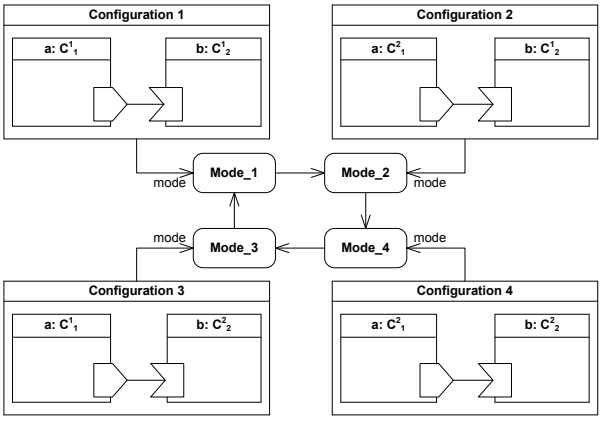


FIGURE 4.8 – Représentation du comportement adaptatif d'un système

Chaque expression groupe les conditions d'utilisation d'une variante utilisée dans la configuration cible de la transition. Pour chaque transition multiple (qui nécessite le déclenchement de plusieurs *triggers* pour son activation) une garde (*guard*) est définie dont sa contrainte exige l'activation de tous les déclencheurs pour exécuter l'effet (*effect*) de la transition. L'effet d'une transition est représenté dans une activité par un *CallOperationAction* qui appelle les opérations de reconfiguration correspondantes à la transition avec les bons paramètres. Par exemple, la figure 4.9 présente l'ajout de l'opération de reconfiguration *Change* dans une activité sous la forme d'un *CallOperationAction*. Le passage entre *Mode_2* et *Mode_4* consiste à utiliser la variante C_2^2 à la place de C_2^1.

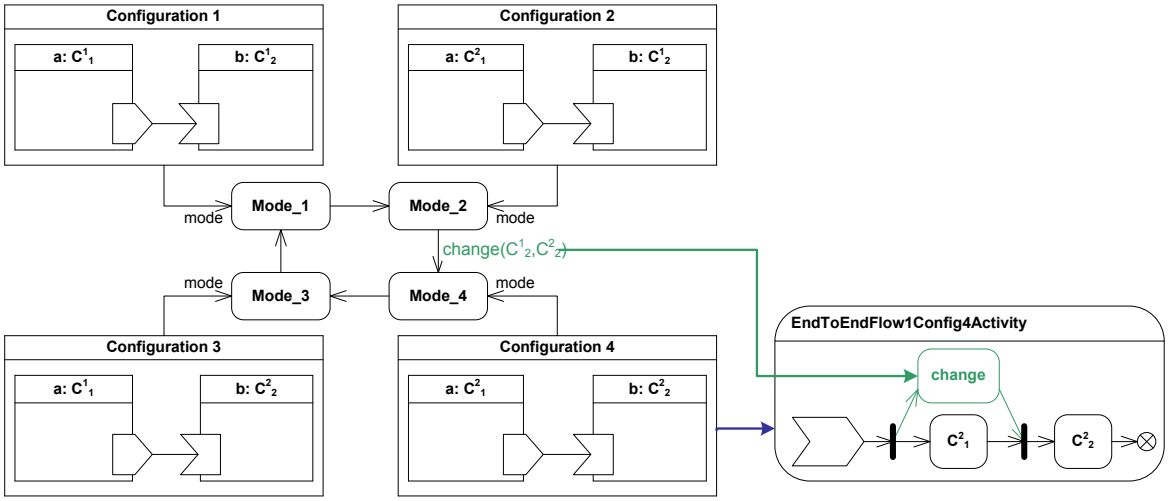


FIGURE 4.9 – L'effet d'une transition d'un comportement adaptatif.

Dans notre approche nous avons opté pour une approche dérivée, c.-à-d. que nous dérivons les configurations à partir des variantes. Ce choix présente des avantages et des inconvénients. L'avantage de cette solution c'est qu'elle donne des configurations optimales. L'inconvénient est qu'elle peut causer une explosion combinatoire de nombre de configurations déclenchée par l'ajout des variantes dans le système. Cependant, ce nombre peut être réduit par l'ajout des conditions d'utilisation aux variantes.

Cela présente une manière d'utiliser le concept de *ModalBehavior* de MARTE pour exprimer le comportement adaptatif. Dans ce qui suit, nous présentons deux méthodes pour générer ce comportement adaptatif dans une approche à base de composants.

4.3 L'automatisation de la création du modèle de comportement adaptatif

Le modèle de comportement adaptatif peut être créé manuellement, généré ou partiellement généré. Dans le cas du modèle généré, les modes sont calculés en utilisant les informations des variantes des composants du système et leurs conditions d'utilisation. En d'autres termes, les modes sont modélisés implicitement. Dans le cas du modèle partiellement généré, les modes sont modélisés explicitement ensuite les variantes sont attachées aux modes.

4.3.1 Comportement adaptatif généré

Le but de la génération est de produire tous les modes possibles du système qui seront utilisés durant l'exécution. Lors de la génération des modes, le contexte d'utilisation et la configuration sont générés progressivement pour chaque mode. Ensuite, nous calculons les transitions possibles entre les modes et nous spécifions les déclencheurs et les effets, le déclencheur est lié au changement de certaines variables de l'environnement et l'effet représente l'opération de reconfiguration. Les éléments générés seront stéréotypés par *ModalBehavior* de MARTE.

En fait, le modèle du comportement d'adaptation est une machine à états finis où :
- **Etat (vertex / nœud)** indique un mode du système et sa configuration et son contexte.
- **La transition** indique la possibilité de basculement entre deux modes.
- **Le déclencheur** d'une transition indique les événements de l'environnement (le changement d'une variable de contexte) qui déclenche la transition de mode.
- **L'effet** d'une transition indique l'ensemble des opérations de reconfiguration dynamique qui réalisent le passage de la configuration du mode source à la configuration du mode cible.

Nous supposons que chaque composant dans le modèle du système a au moins une implémentation. Le modèle abstrait du système est un ensemble de composants interconnectés. La configuration du système pour un contexte spécifique est définie par la sélection d'une implémentation spécifique pour chaque composant du système, c'est-à-dire un ensemble d'implémentations interconnectées pour ce contexte plus une allocation aux ressources matérielles. Le contexte est un ensemble des valeurs ou d'intervalles pour les variables d'environnement. La génération de la machine à état d'adaptation commence par la génération de tous les modes possibles, les configurations et les conditions, ensuite la génération des transitions.

Le *modèle abstrait du système* est composé de l'interconnexion des composants abstraits $(c_1, c_2, ..., c_n)$, ces composants peuvent être des éléments variables ou invariables dans le système. Si le composant (c_i) est un élément variable, il peut utiliser différentes implémentations $(c_i^1, c_i^2, ..., c_i^p)$ à l'exécution. Si le composant (c_j) est un élément invariable, il utilise uniquement l'implémentation par défaut (c_j^0) lors de l'exécution.

4.3.1.1 Génération des modes

Afin de générer tous les modes possibles du système, l'idée est de créer un arbre dont les feuilles représentent ces modes. Chaque nœud de l'arbre possède deux types d'informations :
- Les implémentations qui constituent la configuration (le carré solide, cf. figure 4.10),
- Les conditions d'utilisation de chaque implémentation sélectionnée dans la configuration (le carré en pointillés, cf. figure 4.10). Les conditions d'utilisation d'une configuration sont l'ensemble des conditions des implémentations utilisées dans cette configuration.

La génération démarre par un nœud initial vide, dans lequel les listes de conditions et d'implémentations sont vides. Ensuite, elle ajoute un nouvel ensemble de nœuds fils au nœud initial de l'arbre si la profondeur est égale à zéro, autrement l'ensemble est ajouté

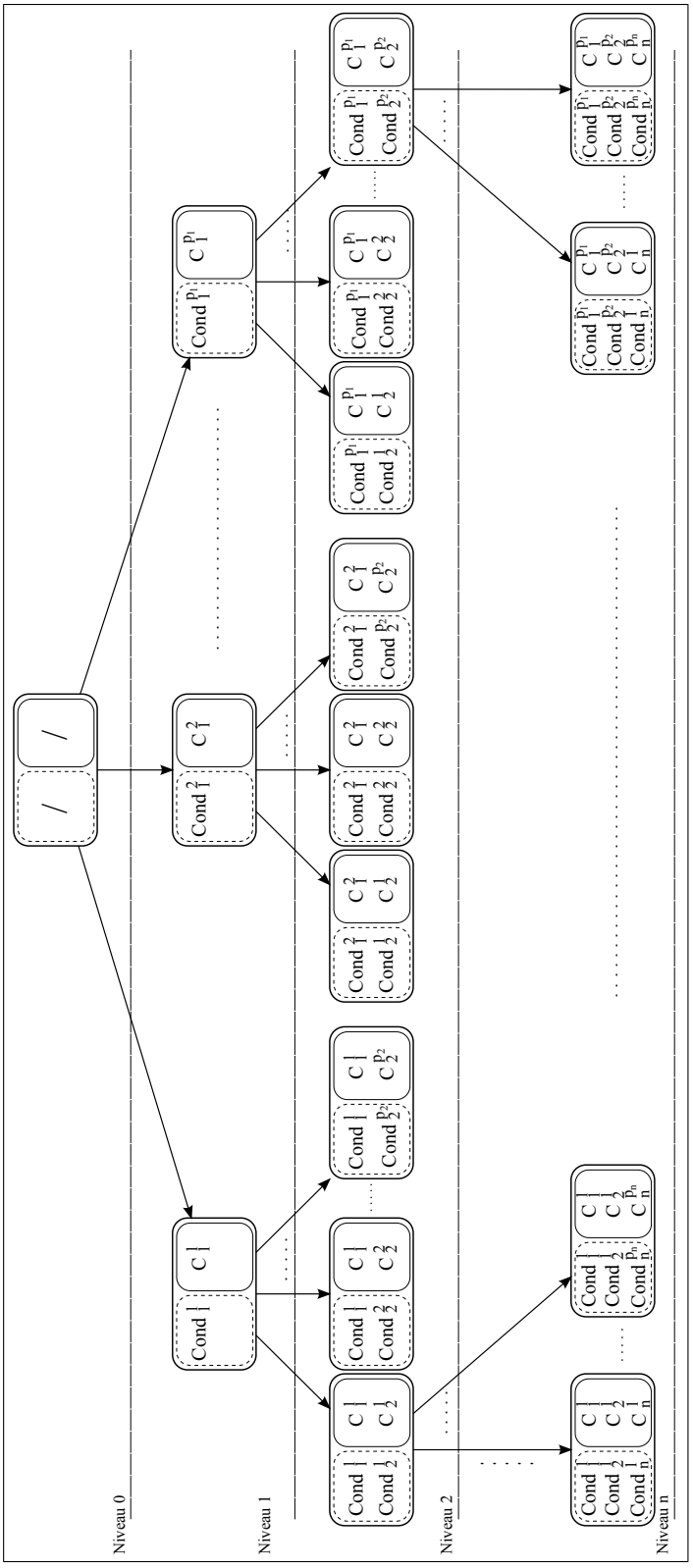


FIGURE 4.10 – Arbre de génération de modes, où :

c_1^1 = l'implémentation $n^\circ 1$ du composant $n^\circ 1$

$cond_1^1$ = la condition d'utilisation de l'implémentation c_1^1

à chaque feuille de l'arbre (si la profondeur d'arbre > 0). La taille de ce nouvel ensemble de nœuds est le nombre d'implémentations (n implémentations : $c_1^1, c_1^2, ..., c_1^p$, cf. figure 4.10) d'un élément variable sélectionné (c_1) aléatoirement parmi les éléments variables du système non traités. Quand les implémentations d'un élément variable sont ajoutées aux feuilles de l'arbre, l'élément est marqué comme traité. Chaque nœud de ce nouvel ensemble contient une implémentation de l'élément variable sélectionné et hérite toutes les implémentations de son nœud parent. Dans la première étape de la génération, les listes de nœud parent (initial, profondeur $= 0$) sont vides, comme le montre la figure 4.10. Les implémentations de l'élément variable c_2 sont ajoutées par la même manière au premier niveau de l'arbre afin d'avoir le deuxième niveau de l'arbre. Quand une implémentation est ajoutée au mode (nœud d'arbre), ses conditions sont également ajoutées s'ils ne sont pas en contradiction avec les conditions existantes dans le mode. Lorsque ces opérations sont terminées pour chaque élément variable dans le système, les implémentations par défaut de tous les éléments invariables dans le système sont ajoutées à chaque feuille de l'arbre. Au cas où le système est composé uniquement des éléments invariables, en conséquence le comportement adaptatif est constitué d'un seul mode et d'une seule configuration. A ce stade, les feuilles de l'arbre représentent tous les modes possibles (configurations et conditions) pour le système. Si le système est constitué de n composants et chaque composant possède p variantes au maximum, alors le nombre de configurations maximal est égale à p^n configurations.

4.3.1.2 Calcul des transitions possibles

Dans la machine à état d'adaptation deux genres de transitions sont possibles : *transition simple* et *transition multiple*. Une simple transition possède un seul évènement déclencheur et une seule opération de configuration en tant qu'effet. Ce genre de transition ne couvre pas toutes les transitions possibles de la machine à état d'adaptation. Une transition multiple a plus d'un déclencheur et plus d'une opération de reconfiguration en tant qu'effet, comme le montre la figure 4.11.

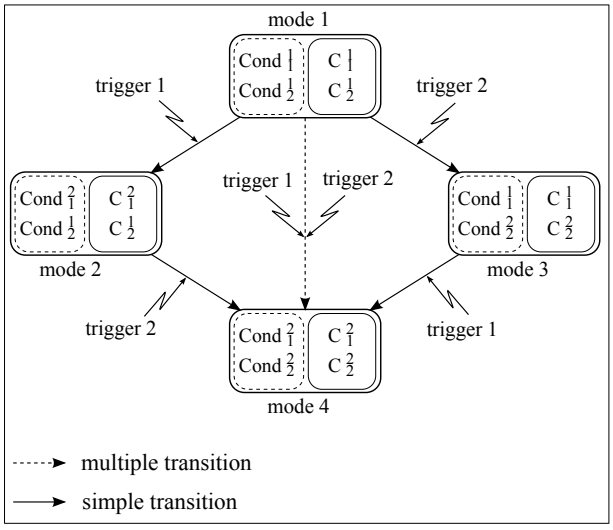


FIGURE 4.11 – Transition simple et transition multiple

L'utilisation des transitions multiples permet de créer un graphe complet (machine à état). La transition simple peut être utilisée seulement entre deux modes dans lesquels les configurations sont différenciées par un seul changement d'implémentation d'un composant. Cependant la transition entre deux modes peut déclencher plus d'un changement d'implémentation. Le déclencheur (*trigger*) d'une transition est activé par la réception d'un évènement de changement (*changeEvent*). Ce dernier est activé lorsque la valeur de son expression de changement (*changeExpression*) passe de *false* à *true*. Cette expression de changement est basée sur une condition d'utilisation d'une des variantes des composants du système. L'utilisation des transitions multiples permet de gagner du temps durant les opérations de reconfiguration dynamique : dans le cas de plusieurs transitions simples, le système doit attendre l'état sûr de chaque transition avant de pouvoir commencer les opérations de reconfiguration. En cas de transitions multiples, le système attend une seule fois pour l'état sûr et puis exécute toutes les opérations de reconfiguration dynamique. L'état sûr pour une opération de reconfiguration est défini comme une situation dans laquelle, toutes les instances d'un composant (sujet de reconfiguration) sont temporairement inutilisées ou ne traitent aucune requête [66]. Une politique d'optimisation de temps de passage entre les configurations est possible, elle peut être réalisée par l'acceptation de nouvelles demandes de reconfiguration entre le premier déclencheur de reconfiguration et son état sûr, ce qui permet la réalisation d'une transition multiple au lieu de plusieurs transitions simples.

Afin de calculer toutes les transitions possibles de la machine à état d'adaptation, une table de transition d'état (STT : *state transition table*) est utilisée avec quelques modifications. La *state transition table* est modifiée pour être en mesure de représenter des transitions multiples, où les intersections ligne/colonne contiennent une liste des paires déclencheur/effet au lieu d'un simple déclencheur/effet, cf. tableau 4.1.

Current \ Next	mode1	mode2	mode3	mode4
mode1	-	t_1/e_1	t_2/e_2	$t_1, t_2/e_1, e_2$
mode2	-	-	-	t_2/e_2
mode3	-	-	-	t_1/e_1
mode4	-	-	-	-

TABLE 4.1 – Table de transition d'état de la machine à état d'adaptation de la figure 4.11, où :

t_x = transition x
e_x = effect x
mode$x$ = state x

Au début du calcul, la *state transition table* est vide. Les calculs de transition sont effectués lors d'une traversée de la *state transition table*. Afin de trouver les éléments déclencheurs pour chaque transition, des tests sont effectués sur les modes Actuels/Suivants de chaque case de la *state transition table*. Un déclencheur/effet est ajouté à la transition, si trois conditions sont satisfaites :

1. La source de la condition (implémentation) de l'état actuel est le même que la source de la condition de l'état suivant.

2. La condition de l'état actuel est sur la même variable d'environnement que la condition de l'état suivant.
3. Soit le type de la condition de l'état actuel n'est pas le même que celui de la condition de l'état suivant. Soit le type de la condition de l'état actuel est le même que celui de la condition de l'état suivant, mais les valeurs des conditions sont différentes.

Ces trois tests sont effectués entre toutes les conditions de l'état actuel et les conditions de l'état suivant de chaque case de la *state transition table*. L'achèvement des opérations précédentes fournit une machine à état d'adaptation complète.

4.3.1.3 Identification des déclencheurs et des effets

Les conditions préalables d'ajout de déclencheur/effet à une transition permettent :
– L'identification de l'élément de l'environnement source de l'événement d'activation d'un déclencheur de la transition.
– L'identification de l'implémentation qui sera changée entre les deux modes (configurations). Par conséquent, le type d'opération (effet) à accomplir pour effectuer la transition entre les deux modes est également identifié.

4.3.2 Comportement adaptatif généré partiellement

En ce qui regarde le comportement adaptatif généré partiellement, les modes et les transitions ne sont pas générés, uniquement les configurations, les déclencheurs et les effets sont générés. Le développeur définit chaque mode et son contexte d'utilisation (la condition sur les valeurs d'une variable d'environnement). Pour chaque variante dans le système, le développeur définit aussi le mode dans lequel la variante peut être utilisée. En utilisant les informations correspondantes aux modes et aux variantes, nous générons une configuration pour chaque mode. La génération d'une configuration d'un mode est basée sur la sélection de la variante destinée à ce mode, cela est fait pour chaque composant dans le système. Les transitions entre les modes sont produites de la même manière de la solution précédente. Le comportement adaptatif généré partiellement est une approche dite descendante (top-down), où l'attention du développeur est focalisée sur les modes. Autrement dit, le développeur définit le contexte d'utilisation de chaque mode et non pas celui des variantes. Cette approche est utile quand les modes du système sont fixés explicitement par la nature du système développé.

Dans ce qui suit, nous présentons la mise en œuvre des deux solutions.

4.4 Mise en œuvre de l'approche

Dans cette section, nous présentons une mise en œuvre des deux solutions proposées. A cet égard, nous présentons trois stéréotypes pour la modélisation des règles d'adaptation, les éléments de contexte et la variabilité dans le modèle du système. Nous présentons également l'utilisation de *state transition table* et le modèle de décision pour la génération de la machine à état d'adaptation.

4.4.1 Modélisation

Pour spécifier les éléments variables dans le modèle de base du système et les règles d'adaptation, un profil avec quatre stéréotypes est défini, cf. figure 4.12. Le premier stéréotype *VariableElement* est utilisé pour spécifier les éléments systèmes qui peuvent utiliser plus d'une variante d'implémentation durant l'exécution. Le deuxième stéréotype *VariantConditions* est utilisé pour indiquer les conditions d'utilisation d'une variante. Chaque condition est spécifiée par le stéréotype *Condition*, qui est appliqué sur l'élément UML *Constraint*. Les conditions de la même variante peuvent être liées par deux types de relation *And* et *Or*.

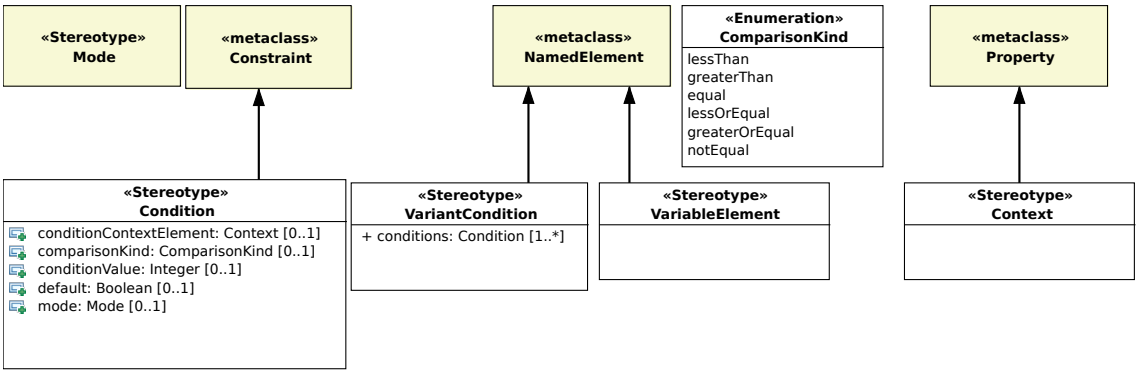


FIGURE 4.12 – Les stéréotypes *VariantConditions*, *Condition*, *VariableElement* et *Context*

Ce stéréotype *Condition* spécifie le contexte d'utilisation d'une implémentation par une comparaison (règle d'adaptation) entre un élément de l'environnement et à une valeur donnée par l'utilisateur. Ce stéréotype est composé de cinq attributs :

1. ***conditionContextElement*** représente l'élément de l'environnement sur lequel la condition est basée (c'est un élément stéréotypé *Context*).
2. ***comparisonKind*** représente le type de comparaison utilisée par la condition, parmi les types définis dans l'énumération *ComparisonKind*.
3. ***conditionValue*** représente la valeur de comparaison, cette valeur est spécifiée par le développeur. Le type de cette valeur est *valueSpecification*.
4. ***default*** un indicateur booléen qui prends *true* comme valeur lorsque le composant n'est pas stéréotypés *VariableElement* et possède plus d'une implémentation, afin d'indiquer que cette variante sera utilisée comme variante par défaut.
5. ***Mode*** est utilisé dans le cas de la génération partielle du comportement adaptatif, pour indiquer le mode dans lequel nous pouvons utiliser la variante. Dans le cas de la génération complète du comportement adaptatif, le mode n'est indiqué, d'où la multiplicité 0..1.

Le stéréotype est *Context* est utilisé pour spécifier les attributs qui représentent les éléments de l'environnement référencé par l'attribut *conditionContextElement* du stéréotype *Condition*, cf. l'exemple du chapitre 6.

Les éléments de la machine à état sont stéréotypés par les stéréotypes *Mode*, *ModeTransition* et *Configuration* de *ModalBehavior* de MARTE.

4.4.2 Les divergences possibles entre les conditions des implémentations

Le contexte d'utilisation d'une implémentation peut être défini par le groupement de plusieurs conditions d'utilisation. Une condition est une comparaison d'une variable d'environnement (contexte) avec une valeur, ex. $x_1 > 10$ ce qui donne l'intervalle $]10, \infty[$. Si les deux conditions sont sur des variables différentes cela ne provoque pas de divergence lors du groupement des conditions (ou des intervalles). Par contre, si les deux conditions sont sur la même variable (ex. $cond1 : x_1 < 5$ et $cond_2 : x_1 \geq -15$, cf. figure 4.13) cela entraînent plusieurs cas de divergence entre les conditions lors de leurs groupement. Deux conditions sont groupés par la relation *And* ou la relation *Or*, l'intersection ou l'union respectivement de leurs intervalles. Le nombre de cas possible de groupement est en fonction de nombre de types de comparaison, le nombre des relations possibles entre deux conditions et les valeurs des variables de comparaison des deux conditions. Chaque changement de ces éléments change le résultat du groupement, ce qui complique l'étude de la divergence.

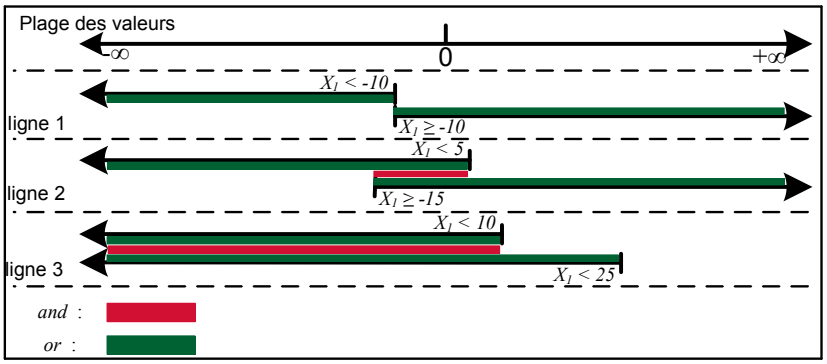


FIGURE 4.13 – Type de divergence entre conditions

Pour faciliter l'étude de la divergence, nous focalisons sur les résultats possibles du groupement et non pas sur les combinaisons possibles pour le groupement. Dans notre approche de création de modèle du comportement adaptatif, les conditions peuvent être groupées pour deux raisons : pour la définition des intervalles d'utilisation d'une variante ou pour la définition du contexte d'utilisation d'une configuration. Dans ce qui suit nous parlons que des conditions qui sont sur la même variable. Le groupement des conditions pour la même variante peut être réalisé par les deux relations *And* et *Or*. Alors que, le groupement des conditions de la même configuration ne se fait que par la relation *And*. Cependant, les conditions des variantes du même élément variable doivent satisfaire d'autres exigences (comme expliquées ci-dessous). Dans cet ordre d'idées, nous présentons le groupement selon les relations *And* et *Or*.

1. Groupement de deux conditions pour la même variante ou la même configuration
   (a) *And* : ce cas figure pour les conditions de la même variante ou la même configuration, cf. le tableau de la figure 4.14.

$i1 \cap i2$	$= \emptyset$	$\neq \emptyset$	$= \emptyset$	$\neq \emptyset$	$= \emptyset$	$\neq \emptyset$	$= \emptyset$	$\neq \emptyset$
$i1 \subset i2$ $= \neg i1 \cap i2$	$= \emptyset$	$= \emptyset$	$\neq \emptyset$	$\neq \emptyset$	$= \emptyset$	$= \emptyset$	$\neq \emptyset$	$\neq \emptyset$
$i2 \subset i1$ $= \neg i2 \cap i1$	$= \emptyset$	$= \emptyset$	$= \emptyset$	$= \emptyset$	$\neq \emptyset$	$\neq \emptyset$	$\neq \emptyset$	$\neq \emptyset$
Résultat de groupement	/	Redondance $i1 = i2$ $\Rightarrow cond1 = cond2$	/	Redondance $i1 \subset i2$ $cond2$ *couvre* $cond1$	/	Redondance $i2 \subset i1$ $cond1$ *couvre* $cond2$	Contradiction	Cas normal
Observation	/	dans ce cas, une des conditions doit être supprimée, parce qu'elle est redondante.	/	dans ce cas, la condition de l'intervalle le plus petit doit être supprimée, parce qu'elle est redondante.	/	dans ce cas, la condition de l'intervalle le plus petit doit être supprimée, parce qu'elle est redondante.	ce cas ne doit pas figurer sur les conditions d'utilisation de la même variante sinon la variante ne sera jamais utilisée par le système du fait que son intervalle d'utilisation et vide. Pour la configuration : ce cas représente un conflit dans la définition de l'environnement d'utilisation de la configuration.	ce type de relation entre les conditions permet d'attribuer plusieurs intervalles d'utilisation à la même variante.

FIGURE 4.14 – Groupement de deux conditions de la même variante par la relation *And*

(b) *Or* : ce cas ne figure que pour les conditions de la même variante, cf. le tableau de la figure 4.15.

$i1 \cup i2$	$= \emptyset$	$\neq \emptyset$	$= \emptyset$	$\neq \emptyset$	$= \emptyset$	$\neq \emptyset$	$= \emptyset$	$\neq \emptyset$
$i1 \subset i2$ $= \neg i1 \cap i2$	$= \emptyset$	$= \emptyset$	$\neq \emptyset$	$\neq \emptyset$	$= \emptyset$	$= \emptyset$	$\neq \emptyset$	$\neq \emptyset$
$i2 \subset i1$ $= \neg i2 \cap i1$	$= \emptyset$	$= \emptyset$	$= \emptyset$	$= \emptyset$	$\neq \emptyset$	$\neq \emptyset$	$\neq \emptyset$	$\neq \emptyset$
Résultat de groupement	/	Redondance $i1 = i2$ $\Rightarrow cond1 = cond2$	/	Redondance $i1 \subset i2$ $cond2\ couvre\ cond1$	/	Redondance $i2 \subset i1$ $cond1\ couvre\ cond2$	/	Cas normal
Observation	/	dans ce cas, une des conditions doit être supprimée, parce qu'elle est redondante.	/	dans ce cas, la condition de l'intervalle le plus petit doit être supprimée, parce qu'elle est redondante.	/	dans ce cas, la condition de l'intervalle le plus petit doit être supprimée, parce qu'elle est redondante.	/	ce type de relation entre les conditions permet d'attribuer plusieurs intervalles d'utilisation à la même variante.

FIGURE 4.15 – Groupement de deux conditions de la même variante par la relation *Or*

2. Exigence entre les conditions des variantes du même élément variable, cf. le tableau de la figure 4.16.
   - Couverture de la plage des valeurs entre les conditions de la même variable : les conditions d'utilisation des variantes d'un élément variable doivent couvrir la plage des valeurs de la variable de la condition. Sinon cela va entrainer l'absence de cet élément variable dans quelques configurations. Ex. une condition de la variante C_1^1 où : $X_1 < 10$ et une condition de la variante C_1^2 où : $X1 \geq -10$, cf. figure 4.13 ligne 1 relation *Or*.
   - L'intersection des intervalles des conditions doit être vide. Autrement, cela provoque un cas d'ambiguïté (contradiction), ex une condition de la variante C_1^1 où : $X1 < 5$ et une condition de la variante C_1^2 où : $X1 \geq -15$, cf. figure 4.13 ligne 2 relation *And*.

   La condition est basée sur le fait que l'intersection doit être vide, donc nous utilisons la relation *And* pour lister tous les cas possible dans le tableau ci-dessous :

$i1 \cap i2$	$= \emptyset$	$\neq \emptyset$	$= \emptyset$	$\neq \emptyset$	$= \emptyset$	$\neq \emptyset$	$= \emptyset$	$\neq \emptyset$
$i1 \subset i2$ $= \neg i1 \cap i2$	$= \emptyset$	$= \emptyset$	$\neq \emptyset$	$\neq \emptyset$	$= \emptyset$	$= \emptyset$	$\neq \emptyset$	$\neq \emptyset$
$i2 \subset i1$ $= \neg i2 \cap i1$	$= \emptyset$	$= \emptyset$	$= \emptyset$	$= \emptyset$	$\neq \emptyset$	$\neq \emptyset$	$\neq \emptyset$	$\neq \emptyset$
Résultat de groupement	/	Contradiction	/	Contradiction	/	Contradiction	Cas normal	Contradiction
Observation	/	Dans ce cas les deux différentes variantes ont la même condition d'utilisation. ce type de relation entre les conditions des variantes du même élément variable représente un conflit. Ce cas ne doit pas figuré entre les conditions dans le système afin de pouvoir générer le comportement adaptatif. Puisque dans ce cas, sur le même intervalle de valeur le système est censé utiliser deux variantes (implémentation) pour le même élément variable (composant) en même temps.	/	Conflit sur l'intervalle d'intersection. $i1 \subset i2$ $cond2$ *couvre* $cond1$. La même explication de la colonne 2	/	Conflit sur l'intervalle d'intersection. $i2 \subset i1$ $cond1$ *couvre* $cond2$ La même explication de la colonne 2	les conditions complémentaires représentent le cas général et recommandé pour les conditions des variantes du même élément variable.	Conflit sur l'intervalle d'intersection. La même explication de la colonne

FIGURE 4.16 – Exigence entre les conditions des variantes du même élément variable

4.4.3 Génération de la machine à état d'adaptation

La machine à état d'adaptation générée est stockée sur un *state transition table* modifiée. La génération des configurations des modes est basée sur le modèle de décision. Dans ce qui suit nous présentons les classes internes du générateur.

Chaque mode est composé de deux listes, comme présenter sur la figure 4.10 de l'arbre de génération de modes. La première liste contient les implémentations utilisées dans la configuration du mode. La deuxième liste contient les conditions d'utilisation de la configuration. La classe *Mode* représente un mode de la machine à état d'adaptation, donc cette classe possède deux attributs, cf. figure 4.17 :

- **conditions** : il s'agit de la liste des conditions d'utilisation de la configuration.
- **configuration** : c'est une liste des implémentations sélectionnées pour un état bien déterminé de l'environnement, elle représente une configuration.

La classe *Part*, représente les informations relatives à une implémentation d'un composant dans une configuration, elle possède 4 attributs :

- **component** : représente la classe abstraite qui modélise un composant applicatif.
- **impl** : la classe d'implémentation qui hérite de la classe abstraite *component*.
- **node** : le nœud sur lequel l'application sera déployée.
- **property** : la propriété utilisée dans le modèle du système pour indiquer la classe abstraite utilisée.

La classe *Condition* représente une condition pour une implémentation, cette classe a presque les mêmes attributs que le stéréotype *Variant*, notamment l'élément de l'environnement, le type et la valeur de la condition, autrement elle définit un attribut de plus :

- **conditionSource** : le type de cet attribut est la classe *Part*, il indique l'implémentation source de la condition.

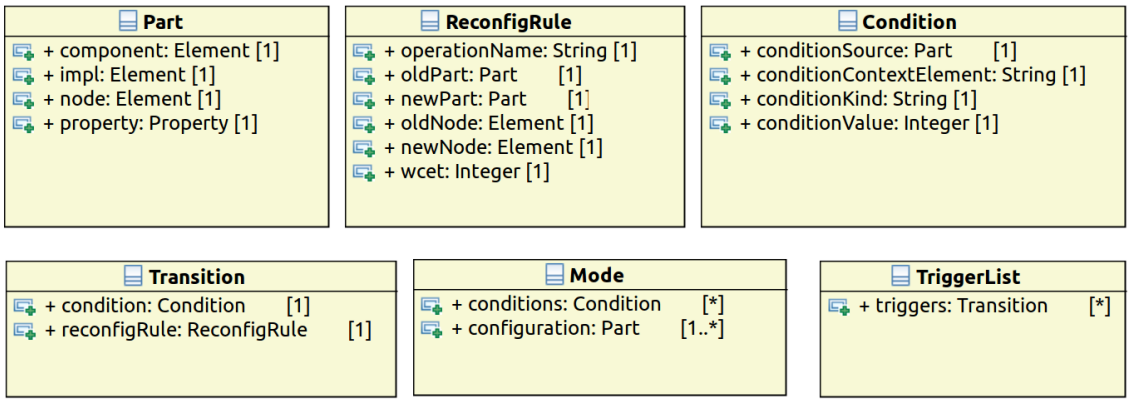


FIGURE 4.17 – Les classes internes du mécanisme de génération de la machine à état d'adaptation

Comme dit précédemment, la machine à état d'adaptation générée est stockée sur un *state transition table*. Les éléments de la STT sont des listes de type *Transition*. La classe *Transition* a deux attributs :

- **condition** : la condition qui contient l'élément de l'environnement déclencheur qui active la transition.
- **reconfigRule** : c'est l'effet de la transition définie par la classe *ReconfigRule*.

La classe *reconfigRule* définit l'effet à travers six attributs :

- **operationName** : l'opération de reconfiguration dynamique effectuée pour réaliser la transition entre les modes (configurations).
- **oldPart** : l'implémentation de la source de configuration qui va subir l'opération de reconfiguration.
- **newPart** : la nouvelle implémentation qui va remplacer l'ancienne implémentation afin de passer à la configuration cible
- **oldNode** : le nœud de déploiement de l'ancienne implémentation.
- **newNode** : le nœud de déploiement de la nouvelle implémentation.
- **wcet** : le pire temps d'exécution de l'opération de reconfiguration.

4.5 Conclusion

Dans ce chapitre, nous avons présenté une approche de modélisation du comportement adaptatif et deux manières de génération du modèle de ce comportement. Cette approche, repose sur le modèle de composant GCM, le concept de *ModalBehavior* de MARTE et le modèle de décision pour la génération. Une fois le modèle du comportement adaptatif complet, nous passons à l'étape de l'analyse de ce modèle vis-à-vis des contraintes temporelles du système présentée dans le chapitre 5. L'analyse des propriétés quantitative des approches et de leurs comparaisons est présentée dans le chapitre 6, où nous présentons également un cas d'étude qui présente en partie l'utilisation de cette solution et son évaluation quantitative.

CHAPITRE 5

L'analyse temporelle du comportement adaptatif des systèmes temps réel

5.1 Introduction

Dans le chapitre précédent, nous avons présenté une approche de modélisation du comportement adaptatif qui repose sur : le modèle de composant *GCM*[1], le concept de *ModalBehavior* de MARTE et deux méthodes de génération du modèle de ce comportement. Dans ce chapitre, nous présentons deux méthodes de validation des caractéristiques temporelles du système vis-à-vis du comportement adaptatif de ce dernier. Chaque méthode est basée sur l'un des paradigmes de modélisation des composants *GCM* de MARTE, notamment la modélisation par flot de données et la modélisation par requête. Pour la modélisation par flot de données, nous proposons une solution qui permet de prendre en compte les opérations de reconfiguration durant l'analyse d'ordonnançabilité en utilisant les analyseurs existants. En ce qui concerne la modélisation par requête, nous proposons une solution qui valide chaque composant séparément. Cette solution est basée sur le calcul du temps libre entre la fin du calcul d'un service et son échéance (fin de période).

5.2 Aperçu du modèle de composant *GCM*

Le modèle de composant générique (GCM) de MARTE peut avoir deux types de port *FlowPort* et *ClientServerPort*. Le type de port détermine le type de communication, notamment la communication par flot (*Flow*) ou la communication par requête. Le port *ClientServerPort* supporte un paradigme de communication demande/réponse (*request/reply*), appelé aussi modèle de communication client/serveur (*client/server*). Les messages qui circulent entre les *ClientServerPort* représentent des appels des opérations ou des signaux. Les ports *FlowPort* permettent de modéliser la communication par flot de données entre les composants. Un port de flot de données peut être un port d'entrée ou un port de sortie de données. Les messages qui circulent entre les *FlowPort* représentent des éléments de données. Le modèle à composants *GCM* spécifie un modèle de concurrence de haut niveau en utilisant le sous-profil *HLAM*[2] de *MARTE*. Ce modèle de concurrence identifie les composants actifs et passifs. Les composants actifs (*RtUnit*) possèdent des ressources d'exécution. Par contre, les composants protégés/passifs (*PpUnit*) ne possèdent pas des ressources d'exécution. Les requêtes reçues par ces composants sont exécutées sur le thread appelant, tout en protégeant l'accès concurrent. Chaque type de communication et catégorie de composant possède un modèle d'exécution particulier. D'ailleurs, le modèle d'exécution influence l'analyse d'ordonnançabilité. Dans ce qui suit, nous présentons pour chaque type de port une solution d'analyse temporelle du système pour son comportement adaptatif. En d'autres termes, nous effectuons une analyse de l'ordonnançabilité du système avec la prise en compte de la charge temporelle des opérations de reconfiguration. Ces solutions sont basées sur la mise en œuvre de *GCM* proposée par eC3M [65].

1. Generic Component Model
2. High Level Application Modeling

5.3 Paradigme de communication flot de données

Un programme/modèle de flot de données est un graphe orienté dans lequel chaque nœud (acteur) représente une fonction et chaque arc orienté un support conceptuel sur lequel les éléments (jetons) de données coulent du producteur vers les consommateurs, cf. section 2.3.2. La communication par flot de données peut avoir deux sémantiques *push* ou *pull*, qui correspondent à deux manières différentes de consommer les données : soit l'attente-active (*polling*) ou en étant appelé à chaque fois de nouvelles données arrivent (*push*). Dans le premier cas, le consommateur utilise son propre thread pour récupérer les données, dans le deuxième cas l'opération *push* est exécutée soit par le middleware ou le thread du producteur de données. Dans les deux cas, le traitement est exécuté sur le thread du consommateur. Nous allons utiliser dans cette partie la sémantique *push*, vu que cela support l'utilisation des techniques de reconfiguration orientées composant (safe state) en exécution, car dans ce cas le thread du composant ne sera activé que lors du traitement de données reçues, c.-à-d. le thread est dormant avant le *push* et réveillé immédiatement après. Contrairement au cas de *pull* qui ne supporte pas de savoir si le composant est en attente-active ou en traitement de données reçues. Dans cette solution, nous supposons que les opérations de reconfiguration dynamique sont plus prioritaires que toutes les opérations (applicatives) du système développé.

L'idée de cette solution est d'effectuer l'analyse d'ordonnançabilité pour chaque configuration ($config_x$) du système (les configurations qui vont être utilisées en exécution) avec la prise en compte des opérations de reconfiguration ($change(B_1, B_2)$) qui seront exécutées afin de passer d'une autre configuration ($config_y$) à celle-ci ($config_x$), cf. figure 5.1.

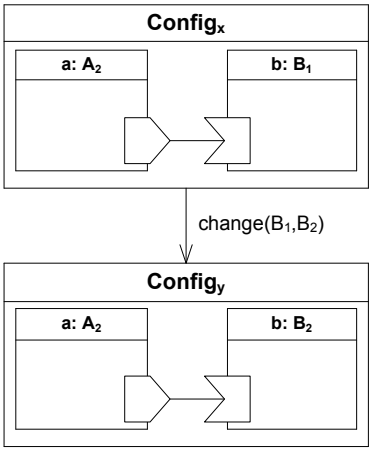


FIGURE 5.1 – Le passage entre deux configurations

Afin d'arriver à cet objectif, nous générons un modèle d'activité pour chaque configuration de notre système. Si ces diagrammes d'activité sont modélisés par le développeur, ce qui est une tâche lourde pour un grand nombre de configuration, ils doivent respecter le mapping présenté par le tableau 5.1 afin de pouvoir générer ses end-to-end flows. Ensuite pour chaque transition nous rajoutons ces opérations de reconfiguration au diagramme d'activité de la configuration (cible de la transition), ensuite nous utilisons ce

dernier modèle d'activité pour calculer les scénarios possibles d'exécution, ces scénarios peuvent être une entrée pour différents analyseurs d'ordonnançabilité (ex. Optimum, MAST, RtDruid, ...), dans ce travail nous utilisons l'analyseur Optimum.

Pour pouvoir générer le diagramme d'activité de chaque configuration ainsi que ses possibles end-to-end flows, le développeur doit modéliser le diagramme d'activité de chaque variante (implémentation) de composant dans le système. Les activités des variantes sont des activités de flow de données composées des nœuds d'action (*action node*) conforme à la définition de *Synchronous data flow* (cf. section 2.3.2) avec une multiplicité de 1. Nous avons utilisé ces conditions sur les nœuds d'actions pour pouvoir générer les end-to-end flows. Autrement, il faut parcourir le comportement défini par le nœud d'action afin de savoir s'il est nécessaire de mettre des données dans tous ses *input pins* pour activer son fonctionnement ou non. Cette tâche peut être imbriquée jusqu'à l'infini, ce qui rend la vérification très difficile. Une variante d'un composant est atomique s'il faut mettre des données dans tous ses *incoming Activity Parameter Nodes* afin d'activer son fonctionnement et produire des données dans ses *outgoing Activity Parameter Nodes*, autrement la variante est hiérarchique.

Afin de détailler nous prenons un exemple d'un système composé de deux propriétés (property) a et b. La propriété a est typée par le composant abstrait A qui dispose de deux implémentations A_1 et A_2. De la même manière, la propriété b est typée par le composant B qui dispose de deux implémentations B_1 et B_2. Le diagramme d'activité de chaque implémentation dans le système est défini, cf. figure 5.2.

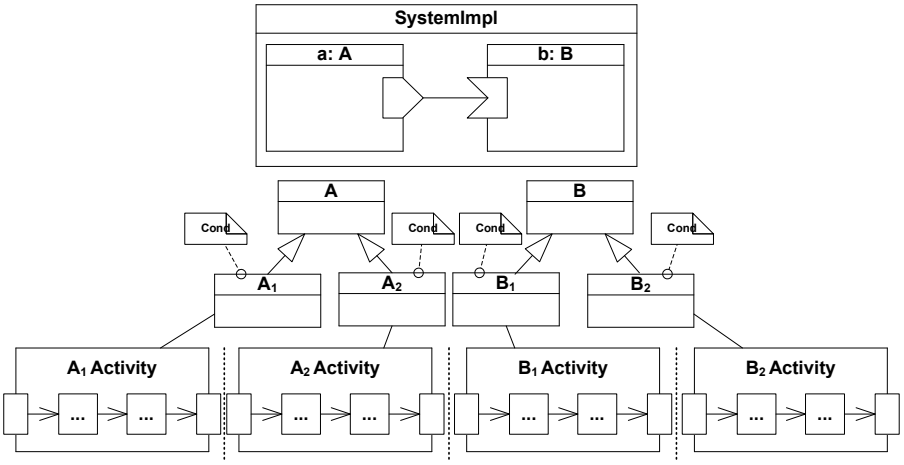


FIGURE 5.2 – Les variantes des composants et leurs diagrammes d'activité.

L'utilisation du générateur de comportement adaptatif présenté dans le chapitre précédent pour cet exemple, nous donne une machine à état d'adaptation (simplifiée dans la figure 5.3) constituée de quatre modes. Pour la simplicité de présentation, nous illustrons les configurations et les modes.

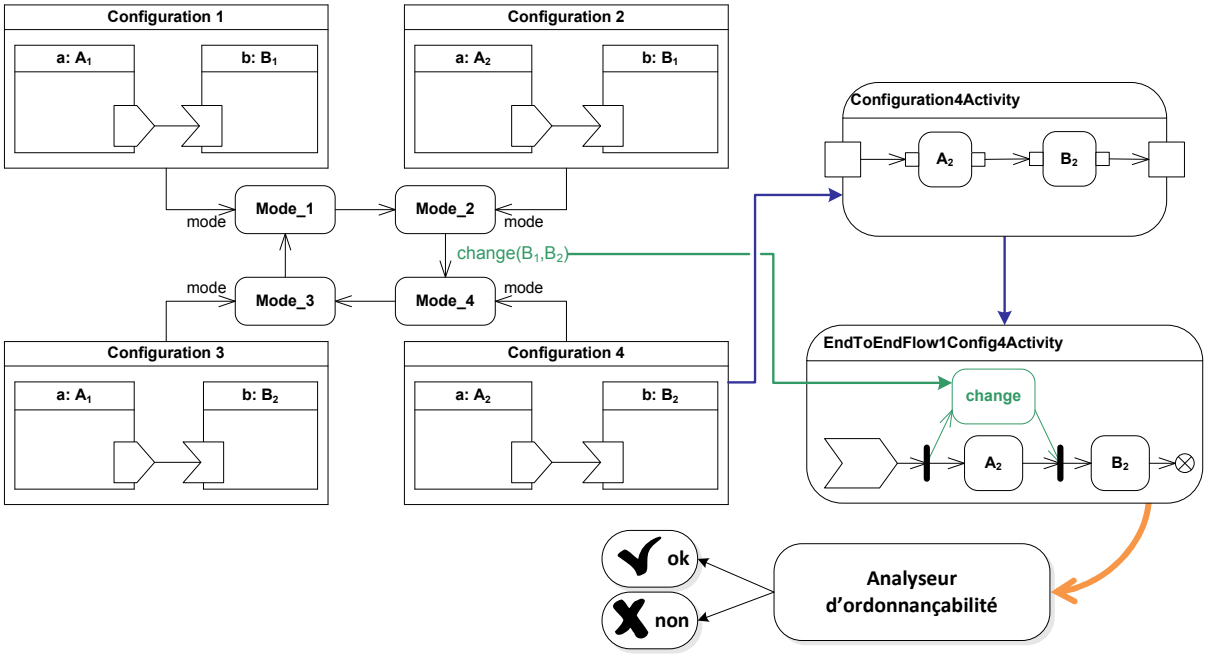


FIGURE 5.3 – La machine à état d'adaptation et l'analyse d'ordonnançabilité

Dans cette solution, nous générons le modèle de comportement pour chaque configuration de la machine à état dans le formalisme d'activité (cf. figure 5.3, configuration 4). À la fin de cette étape, nous parcourons toutes les transitions de la machine à état et nous générons tous les scénarios possibles (end-to-end flow) pour ce modèle d'activité de la configuration cible de la transition. Pour chaque transition, nous ajoutons ces opérations de reconfigurations au bon endroit dans tous les end-to-end flows de l'activité de sa configuration cible. Chaque opération de reconfiguration est ajoutée avant le *CallBehaviorAction* de son composant sujet de reconfiguration. Par exemple, dans la figure 5.3 le *CallBehaviorAction* de l'opération de reconfiguration $change$ (en couleur verte) est ajoutée avant le *CallBehaviorAction* B_2 du composant B_2 sujet de l'opération de reconfiguration ($change$) durant le passage de la configuration 2 à la configuration 4. Cette opération de reconfiguration ($change$) peut être exécutée en parallèle avec toutes les opérations (A_2) qui précèdent son sujet de reconfiguration B_2 parce qu'elle est indépendante de ces opérations. Ensuite, les scénarios seront analysés par un analyseur d'ordonnançabilité. Enfin, les contraintes temporelles du système ne seront pas influencées par son comportement adaptatif si toutes les transitions sont validées (ordonnançable) par l'analyseur d'ordonnançabilité. Ce processus de validation du comportement adaptatif envers les contraintes temporelles est représenté par l'algorithme suivant :

.

Algorithm 1 validation of real-time application constraints considering adaptive behavior

Require: .
$smAdaptation$, the State Machine of Adaptation
$model$, the system model in dataFlow paradigm
Ensure: .
$true$: if the system is validated
$false$: if the system is not valid, and the error source
1: $activityConfigList \leftarrow$ generate the activity model for each configuration,
2: **for** each transition $transition \in smAdaptation$ **do**
3: $activityConfig_i \leftarrow$ the activity of the target configuration of the transition $transition$
4: $endToEndFlowSet_i \leftarrow$ generate all possible end-to-end flows for $activityConfig_i$
5: add the reconfiguration operations of $transition$ to each end-to-end flow $\in$ $endToEndFlowSet_i$
6: **for** each end-to-end flow $endToEndFlow \in endToEndFlowSet_i$ **do**
7: analysing the schedulability of $endToEndFlow$ using existing analyser
8: **if** $endToEndFlow$ is not schedulable **then**
9: log.print(" $endToEndFlow$ is not schedulable for the transition $transition$")
10: **return** $false$
11: **end if**
12: **end for**
13: **end for**
14: **return** $true$

5.3.1 Génération de modèle d'activité à partir d'un modèle composite

Dans cette section, nous présentons les règles de passage d'un modèle composite du système modélisé par le paradigme de flot de données au modèle de comportement dans le formalisme d'activité. Ce comportement est lié à l'écoulement de données entre les composants et pas au comportement interne de chaque composant. Mais avant de présenter ce mapping, nous présentons brièvement le modèle du système modélisé par le paradigme de flot de données en utilisant le port *FlowPort* de MARTE. Dans ce modèle composite chaque port de chaque composant est stéréotypé par le *FlowPort* de MARTE. L'attribue direction du stéréotype *FlowPort* permet de spécifier le sens d'écoulement du flot de données (*in*, *out*, *inout*). Pour la valeur *in*, la direction du flux d'informations va de l'extérieur vers l'intérieur de l'entité propriétaire de port (le composant). Quant à la valeur *out*, la direction du flux d'informations va de l'intérieur vers l'extérieur de l'entité propriétaire. Pour la valeur *inout* le flux d'informations est bidirectionnel. Dans ce travail nous avons utilisé les deux types *in* et *out*. Comme nous présentons sur la figure 5.4, le système est modélisé par une classe principale *SystemImpl* composée des propriétés interconnectées a, b, c, d et e, une pour chaque composant du système A, B, C, D et E respectivement. Chaque propriété est typée par un composant. Nous avons choisi 5 composants afin de pouvoir illustrer tous les cas possibles du mapping.

L'idée du mapping consiste à créer une activité qui représente l'écoulement de données à partir des entrées de la classe système jusqu'à ses sorties en passant par ses éléments

(composants) internes. Pour la classe système, nous créons une activité. Pour chaque port de flot d'entrée de la classe (respectivement port de flot de sortie) nous créons un paramètre d'entrée (respectivement un paramètre de sortie) sur l'activité. Les données associées au message d'entrée sur le port de flot seront traitées comme un jeton sur une *ActivityParameterNode* correspondant au paramètre de l'activité. Pour chaque connecteur de la classe, un flot d'objet (*object flow*) est créé sur l'activité. Ces règles de mapping sont utilisées pour créer l'activité de la classe principale du système (*SystemImpl*). Par contre, les propriétés de la classe principale sont représentées sur son activité par des *CallBehaviorAction*. Les ports d'entrée (respectivement de sortie) du composant type de la propriété sont représentés par des *inputPin* (respectivement *OutputPin*) sur le *CallBehaviorAction*. Le comportement de ce dernier est défini par l'activité de comportement du composant type. La figure 5.4 présente l'activité générée *SystemImplActivity* de la classe *SystemImpl*.

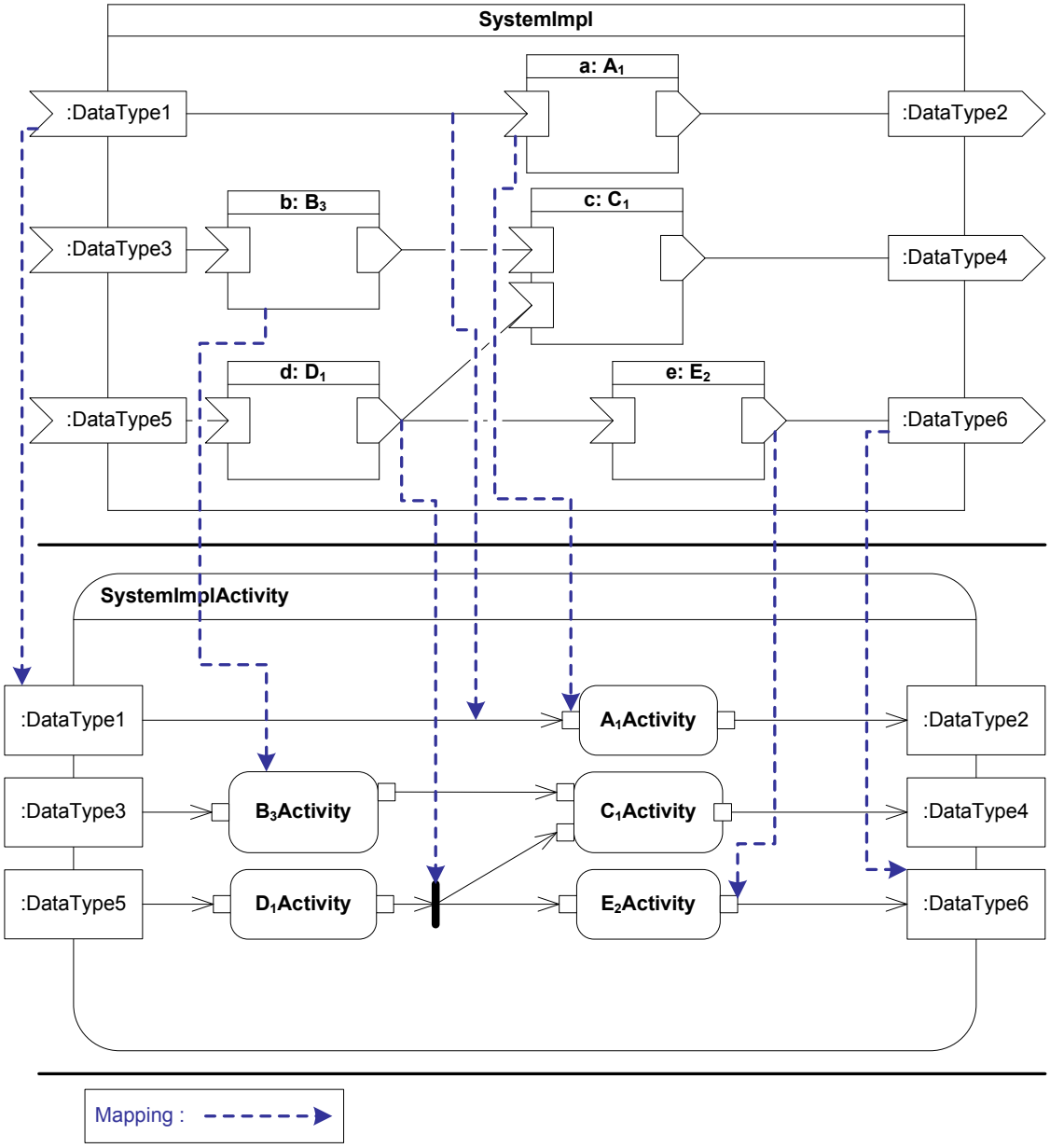


FIGURE 5.4 – Diagramme composite (en *DataFlow*) et son modèle d'activité généré.

Si plusieurs connecteurs sont connectés au même port de sortie, cela est représenté par un *Fork node* dans l'activité. De la même manière, si plusieurs connecteurs sont connectés au même port d'entrée cela est représenté par un *Merge node*. Le tableau 5.1 résume l'ensemble des règles de mapping utilisées afin de générer l'activité du comportement à partir d'un modèle composite non-comportemental :

Composite model	Activity model
class or *component*	*Activity*
In flowPort from the class	Incoming *Activity Parameter Node*
Out flowPort from the class	outgoing *Activity Parameter Node*
Property	*Call behavior action*
The *activity* of the type of the *property*	The *behavior* of the *Call behavior action*
In flowPort from the type of the property	*Input Pin*
Out flowPort from the type of the property	*Output Pin*
Connector	*Object flow*
Multiple connector in an *Out flowPort*	*Fork Node*
Multiple connector in an *In flowPort*	*Merge Node*

TABLE 5.1 – Les règles de passage d'un modèle composite du système modélisé par le paradigme flot de données au modèle de comportement dans le formalisme d'activité.

5.3.2 Génération des scénarios *end-to-end flows*

Le modèle d'activité généré dans la section précédente permet de générer les scénarios possibles (*end-to-end flows*) qui vont servir à l'analyse d'ordonnançabilité du système. Un *end-to-end flow* identifié explicitement un flux de données provenant des capteurs jusqu'à l'environnement externe (actionneurs). Pour chaque *end-to-end flow*, nous pouvons spécifier des contraintes temporelles (ex. le temps de réponse et l'échéance). En vue d'analyser un système, tous ces scénarios (*end-to-end flows*) possibles doivent être analysés. Afin de générer tous les scénarios possibles, nous parcourant l'activité en commençant par un de ces *incoming Activity Parameter Node* et en suivant l'écoulement de flot des jetons (*token*) jusqu'aux *outgoing Activity Parameter Nodes* ; à chaque étape nous créons l'élément correspondant selon la table 5.2.

Activity model	End to end flow
Incoming *Activity Parameter Node*	*Accept Event Action*
outgoing *Activity Parameter Node*	*Flow Final Node*
Call behavior action	outgoing *Call behavior action*
Object flow	*Control Flow*

TABLE 5.2 – Les règles de création des éléments d'un end-to-end flow à partir d'une activité.

Dans cet algorithme, nous avons deux types de parcours : un pour satisfaire les *Input Pin* des *Call behavior action* et l'autre pour activer leurs *Output Pin*. L'algorithme construit

une liste des *incoming Activity Parameter Node (APN)*. Pour chaque élément de cette liste, l'algorithme marque les *Input Pin* des *Call behavior action* connectés à cet *incoming APN* actif. Ces derniers *Call behavior action* sont ajoutés dans une liste pour laquelle nous devons activer les *Output Pins* de chacun de ses éléments après l'activation de tous les *Input Pins* non activés de cet élément. Afin d'activer un *Input Pin*, les *Input Pins* de ces prédécesseurs doivent être tous activés jusqu'aux *incoming APN* nécessaires. Ce dernier sera supprimé de la liste des *incoming APN* à traiter (la liste précédente). Une fois tous les *Input Pins* d'un *Call behavior action* sont activés nous procédons à l'activation de ces *Output Pins*. Activer un *Output Pin* veut dire activer les *Input Pins* de ces successeurs jusqu'aux *outgoing APN* qui seront représentés par des *Flow Final Node*. À la fin du calcul, les opérations de reconfiguration dynamique sont ajoutées aux *end-to-end flow* en tant que *Call Operation Action*. Chaque *end-to-end flow* est modélisé dans une activité. Ce processus est résumé par le pseudo code de l'algorithme 2.

Algorithm 2 the generation of end-to-end flows of an activity

Require: .
act, the activity
$model$, the system model in dataFlow paradigm
$transition$ the transition selected for the analysis
Ensure: .
$endToEndFlowList$, a set of end-to-end flows
1: $incomingAPNList \leftarrow$ the incoming Activity Parameter Nodes of act
2: **while** $incomingAPNList \neq \phi$ **do**
3: $incomingAPNj \leftarrow$ select an incoming Activity Parameter Nodes from $incomingAPNList$
4: $endToEndFlow \leftarrow$ create an activity end-to-end flow for $incomingAPNj$
5: $acceptEventAction \leftarrow$ create an Accept Event Action for $incomingAPNj$ in $endToEndFlow$
6: mark all Input Pin connected to $incomingAPNj$
7: $markedCBAList \leftarrow$ all Call Behavior Action owner of the marked Input Pins
8: **while** $markedCBAList \neq \phi$ **do**
9: $markedCBA \leftarrow$ select Call Behavior Action from $markedCBAList$
10: create a Call Behavior Action in $endToEndFlow$ using $markedCBA$
11: activate and mark all not marked Input Pin of $markedCBA$
remove all $incomingAPNi$ from $incomingAPNList$ reached during the activation and marking
12: activate all Output Pin of $markedCBA$
13: remove $markedCBA$ from $markedCBAList$
14: **if** the successors of $markedCBA$ is an outgoing Activity Parameter Nodes **then**
15: create a Flow Final Node in $endToEndFlow$ using the successors of $markedCBA$
16: **else**
17: add the successors of $markedCBA$ to $markedCBAList$
18: **end if**
19: **end while**
20: remove $incomingAPNj$ from $incomingAPNList$
21: $reconfOpList \leftarrow$ the reconfiguration operations in the effect of the $transition$
22: add the $reconfOpList$ to $endToEndFlow$
23: add $endToEndFlow$ to $endToEndFlowList$
24: **end while**
25: **return** $endToEndFlowList$

Les figures 5.5 et 5.6 présentent les scénarios possibles pour l'activité de la figure 5.4. Nous avons utilisé cet exemple composé de 5 éléments dont l'activité de C_1 est atomique avec deux input pins afin de pouvoir présenter la possibilité de générer un end-to-end flow qui contient plus d'un *AcceptEvent*. Nous supposons que la configuration de l'activité présentée dans la figure 5.4 est une cible d'une transition dont l'effet utilise l'opération de reconfiguration $change$. Cet opération $change(E1, E2)$ est utilisée afin de changer l'implémentation utilisée par la propriété e de $E1$ à $E2$. Dans ce cas, nous ajoutons l'opération de reconfiguration aux *end-to-end flows* qui contient l'activité de $E2$ pour la

prendre en compte durant l'analyse tel que présenté sur les figures 5.5 et 5.6 :

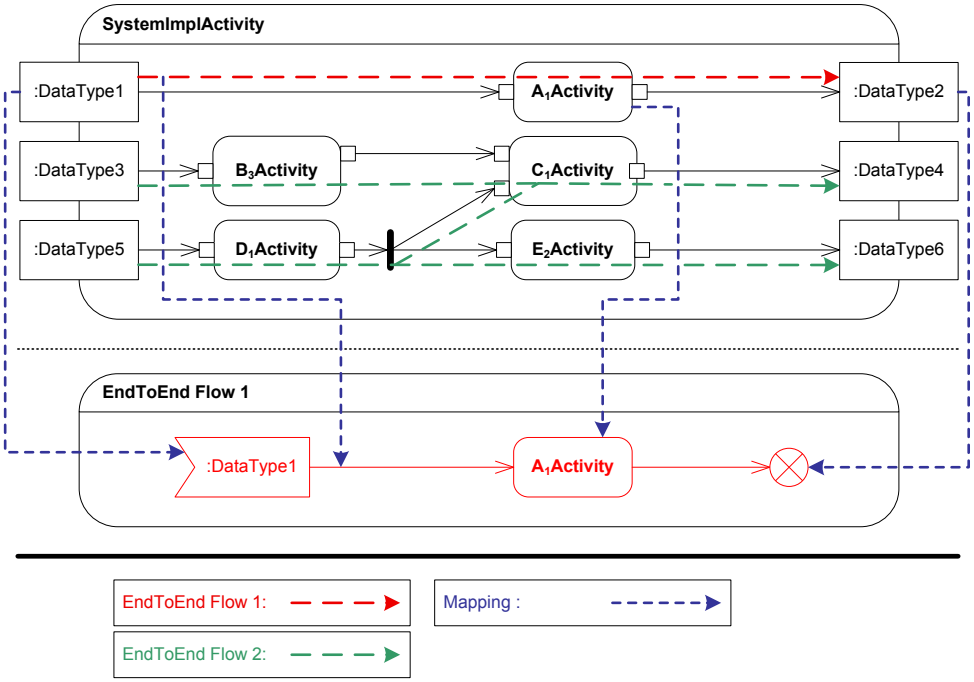


FIGURE 5.5 – Une activité et son premier *end-to-end flow* (scénario).

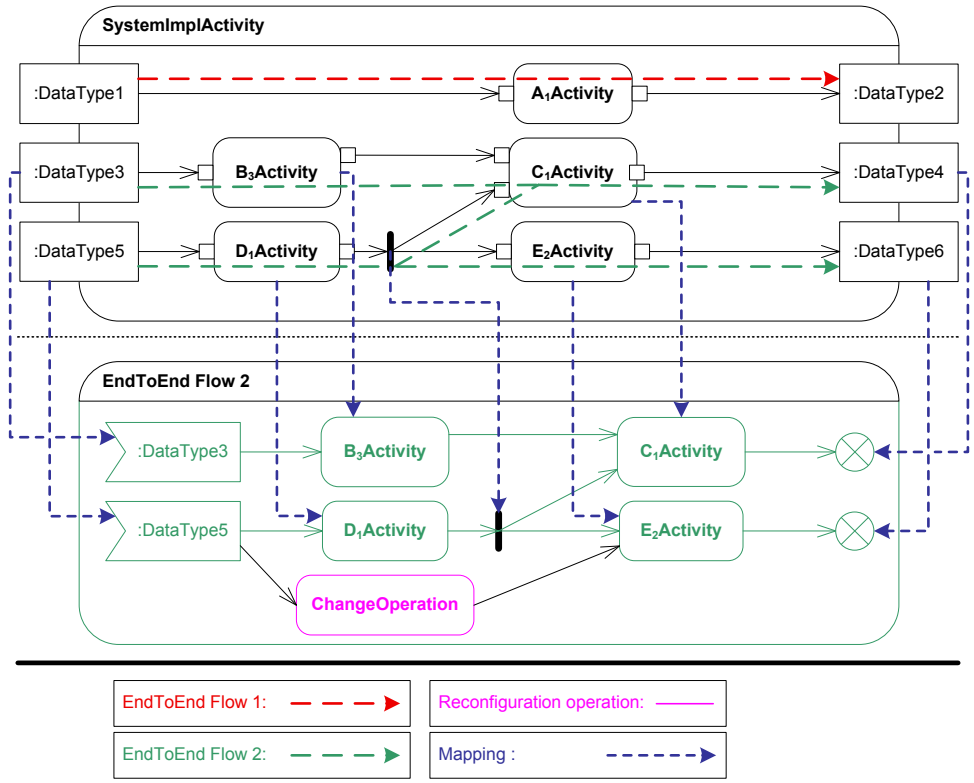


FIGURE 5.6 – Une activité et son deuxième *end-to-end flow* (scénario).

5.3.3 Utilisation d'un analyseur d'ordonnançabilité (Optimum)

Les *end-to-end flows* générés dans la section précédente englobent toutes les informations nécessaires à un analyseur d'ordonnançabilité afin d'effectuer l'analyse. Dans cette étape nous avons utilisé l'analyseur de la méthodologie *Optimum* [22]. Cette méthodologie fait l'interface avec plusieurs analyseurs d'ordonnançabilité, notamment *MAST* et *RT-Druid*. Elle fournit également ses propres implémentations pour quelques algorithmes d'analyse d'ordonnançabilité. Il est possible d'utiliser les *end-to-end flows* générés avec d'autres analyseurs (ex. *Cheddar*). Dans ce travail nous avons choisi *Optimum* du fait qu'il prend en entrée un modèle UML annotée par MARTE. Autrement dit, nous l'avons choisi pour la facilité de son utilisation dans notre contexte.

Les *end-to-end flows* générés pour une activité d'une configuration représentent le modèle *Workload Behavior* de cette configuration. À partir de ce modèle de *Workload Behavior*, des informations de la plateforme matérielle et logicielle (tâche), le mapping des *end-to-end flows* sur les tâches et des tâches sur les processeurs, *Optimum* nous fournit les résultats de l'analyse d'ordonnançabilité, notamment celle de chaque tâche et son temps de réponse. Pour plus de détails sur le fonctionnement d'*Optimum* voir [22]. Nous fournirons plus de détails dans le chapitre suivant 6, où nous étudions un exemple et nous y appliquons les techniques de ce chapitre, i.e. la génération des end-to-end flows et leur analyse.

5.4 Paradigme de communication client-serveur

Une analyse d'ordonnançabilité d'un modèle orienté objet n'est pas faisable, étant donné que ce dernier est dans un haut niveau d'abstraction (classe, appel d'opération, héritage, encapsulation, etc.), alors que l'analyse d'ordonnançabilité utilise des éléments d'un niveau plus concret liés au modèle d'exécution (tâche, thread, relation de dépendance et de précédence entre tâches, etc.). Afin d'analyser un modèle orienté objet il faut :

- Restreindre le modèle d'exécution du paradigme orienté objet (ex. ne pas autoriser la création dynamique de tâche).
- Fixer un mapping des éléments du modèle orienté objet (objet, message, etc.) sur les threads d'exécution et un déploiement de ces derniers sur les nœuds de calcul.

Les solutions existantes [67, 68, 69, 70, 71] proposent une réduction du modèle orienté objet à un modèle proche au concept de flot de données afin de faciliter l'analyse d'ordonnançabilité. Dans ce cas, nous pouvons utiliser ces solutions d'une façon complémentaire avec notre solution de validation du comportement adaptatif pour le paradigme de flot de données, présentée dans la section précédente.

Dans ce qui suit nous présentons une solution qui permet de valider le comportement adaptatif de chaque composant dans le système séparément. Autrement dit, elle ne vérifie pas l'influence du comportement adaptatif d'un composant sur un autre composant. Cette solution est nécessaire pour la validation de chaque composant séparément, mais elle est non suffisante pour la validation du système complet. Donc une analyse d'ordonnançabilité est nécessaire afin de valider le système.

La communication par appel d'opération peut être *synchrone* ou *asynchrone*, chaque cas conduit à un modèle d'exécution spécifique. Nous allons utiliser dans cette partie l'appel d'opération *synchrone*, vu que son modèle d'exécution permet d'utiliser les techniques

de reconfiguration orientées composant (*safe state*) en exécution. Cette limitation est liée au modèle d'exécution et non à l'aspect de communication *asynchrone* (plus de détails ultérieurement).

Dans cette solution, nous supposons que les opérations de reconfiguration dynamique sont plus prioritaires que toutes les opérations (applicatives) du système développé. Nous supposons aussi que le pire temps d'exécution (*WCET*) de chaque élément du système est calculé au préalable.

L'interaction (communication) dans cette solution est basée sur le type de port *ClientServerPort* synchrone de *GCM*, de son modèle d'exécution et de sa mise en œuvre proposée par eC3M. Ce modèle d'exécution utilise un thread pour chaque composant et un thread pour chaque service fourni par le composant, ce dernier thread est créé lors de l'appel de son service. Le composant 2 de la figure 5.7 démarre un thread pour chaque service appelé. Nous avons opté pour ce modèle d'exécution pour faciliter l'utilisation des techniques de la reconfiguration dynamique, spécialement pour l'atteinte de l'état sûr (*Safe state*) d'un composant. Cet état sûr est basé sur le nombre de threads en cours d'exécution pour le compte de ce composant.

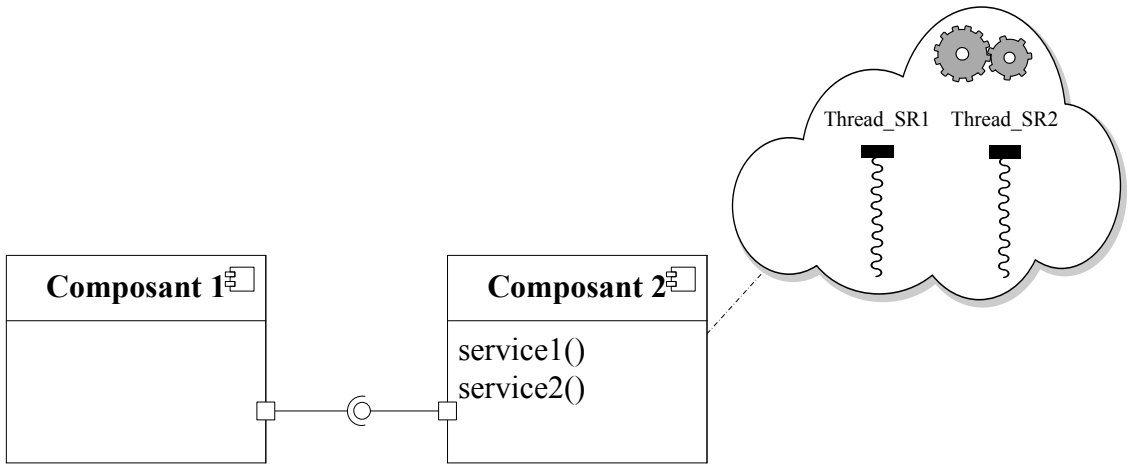


FIGURE 5.7 – Le modèle d'exécution pour une communication client/serveur synchrone

Notre travail est basé sur le modèle de tâche temps réel de MARTE. Le *RealTimeFeature* de MARTE (section F.7.9) définit des caractéristiques qui peuvent être attachées par l'utilisation des stéréotypes *RtFeature* et *RtSpecification* à un service temps réel (tâche), un message ou un signal, dont cinq qui nous intéressent sont :

- **OccKind** : la nature d'occurrence de la tâche (*task occurrence kind*) représente le modèle d'arrivée de la tâche, par exemple périodique, apériodique.
- **tRef** : le temps de référence utilisé par les autres attributs relatifs au temps.
- **relDl** : échéance (*deadline*) relative par rapport au temps de référence, représente un délai.
- **AbsDl** : échéance (*deadline*) absolue.
- **rdTime** : le temps minimal pour être prêt à l'exécution (*ready-time*), l'exécution ne devrait pas commencer avant cette durée.

Nous utilisons un modèle de tâche simplifie dont toutes les tâches sont périodiques avec une échéance relative et égale à la période.

Afin d'illustrer la problématique, l'idée de la solution et sa mise en œuvre, nous utilisons l'exemple avionique de la section 13.4.2 de MARTE. L'exemple présente un cas d'utilisation du sous-profil *HLAM* pour annoter le modèle d'un système de navigation avionique avec les caractéristiques temps réel. La figure 5.8 donne un aperçu de ce modèle. Cette figure présente un diagramme de séquence de l'interaction entre les composants *Trajectory* et *LocationAccess*. Le composant *Trajectory* appelle l'opération *getLocation* du composant *LocationAccess* périodiquement, avec une période de $10ms$.

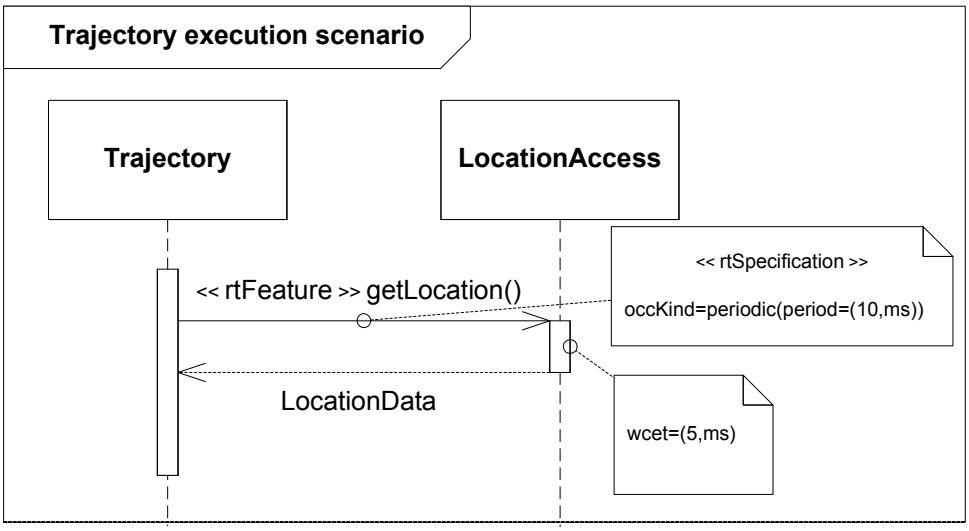
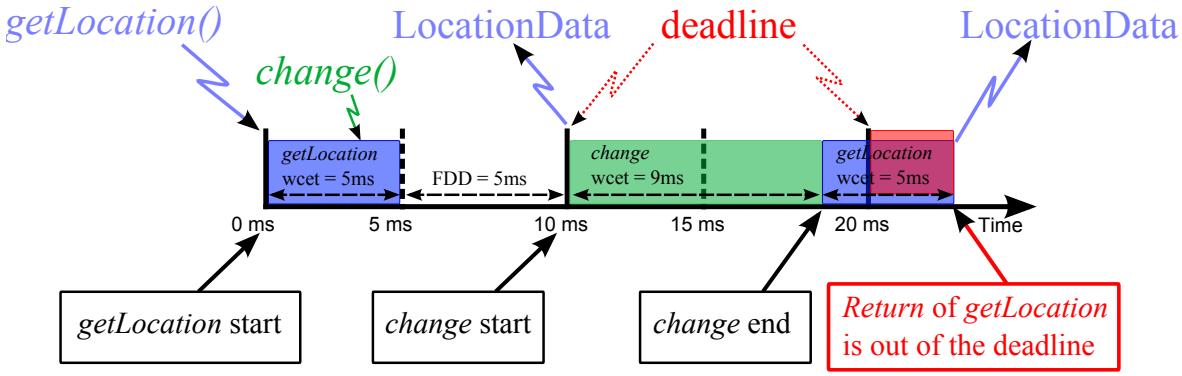


FIGURE 5.8 – Diagramme de séquence du composant *Trajectory*

Nous supposons que le pire temps d'exécution de *getLocation* est de $5ms$ et celui de *change* est de $9ms$. *Change* est une opération de base de la reconfiguration dynamique, elle effectue le changement d'un composant par un autre à l'exécution. Nous supposons également que le composant *LocationAccess* possède deux implémentations différentes. Chaque application est dédiée à un environnement spécifique. Si l'environnement de l'implémentation non utilisée survient, le système exécute une opération de changement (*Change*) afin d'utiliser l'implémentation correspondante. Dans cet exemple, l'opération de changement influencera le respect des contraintes de temps du système, cf. figure 5.9.

Afin de respecter l'échéance de l'opération *getLocation*, son WCET et celui de *Change* doit être inférieure ou égale à sa période (ou son échéance généralement) :

$$getLocationWCET + changeWCET \leq period \text{ ou } Deadline \text{ de } getLocation$$



getLocation démarre l'exécution à $0ms$, à $4ms$ *Change* arrive. Lorsque *getLocation* retourne la valeur à $10ms$ (le composant est en état sûr) l'opération *Change* démarre son exécution. À 19 *Change* termine l'exécution, dans ce cas *getLocation* ne peut pas envoyer la valeur de retour avant l'échéance (20), cela implique une violation d'échéance.

FIGURE 5.9 – L'influence de respect des contraintes de temps du système par une opération de reconfiguration

Pour effectuer cette validation, la durée maximale libre avant l'échéance (FDD : *Free Duration before the Deadline*) doit être calculée pour chaque tâche susceptible de subir une opération de reconfiguration. FDD est l'intervalle de temps entre la fin de l'exécution de la tâche et son échéance qui peut être soit relative ou absolue. Dans le premier cas, le FDD est égal à $absDl - rdTime - wcet\_task$. Pour le deuxième cas, le FDD est égal à $tRef + relDl - rdTime - wcet\_task$. Nous avons : $tRef + relDl = asbDl = d$ donc le FDD est égal à $d - rdTime - wcet\_task$. Pour une raison de simplicité, nous supposons que *rdTime* est le début de l'exécution effective et l'échéance correspond à la fin de la période. Cela implique que : $FDD = period - rdTime - wcet$. Ainsi, la condition deviendra comme suit :

$$\mathbf{FDD \geq wcet_{ro}} \qquad (5.1)$$

Où : $FDD = period - rdTime - wcet_{task}$

$\mathbf{wcet_{ro}}$: le pire temps d'exécution de l'opération de reconfiguration
$\mathbf{wcet_{task}}$: le pire temps d'exécution de la tâche sujet de reconfiguration
period : la période de la tâche sujet de reconfiguration

Par exemple, si le temps *rdTime* de la tâche *getLocation* est de $10ms$, son échéance est de $25ms$ et le pire temps d'exécution est de $7ms$, donc le FDD est égal à $25 - 10 - 7 = 8ms$. L'utilisation de cette condition est possible si chaque composant dans le système développé fournit un seul service non réentrant, parce que l'exécution simultanée de deux instances de la même tâche n'est pas prise en compte par cette formule. Si le service est réentrant, donc sa tâche peut donc avoir deux instances ou plus en exécution simultanée et dans ce cas nous

ne pouvons pas calculer le FDD vu que nous ne connaissons pas le nombre des instances et l'instant de leurs arrivées.

Un service d'un composant peut être appelé par plusieurs composants. Chaque composant appelle le service avec une contrainte de temps différente, cf. figure 5.10 : le *composant 2* appelle le *service 1* du *composant 1* avec une période de $10ms$, le *composant 3* appelle le même service avec une période de $15ms$ et le *composant 4* appel aussi le même service avec une période de $20ms$. Afin de calculer FDD dans ce cas, nous utilisons la période la plus petite, dans l'exemple de la figure 5.10 nous utilisons $10ms$ comme période.

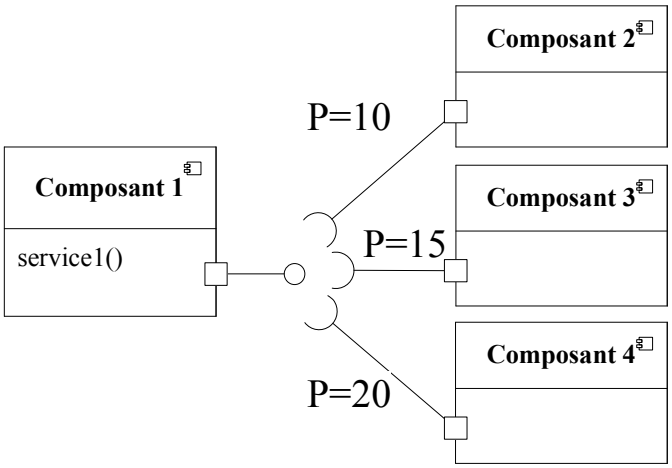


FIGURE 5.10 – Un service réentrant et le choix d'une période pour le calcul de FDD

Cette validation ne sera pas effectuée pour chaque opération de chaque composant du système développé, mais elle sera limitée aux opérations des composants qui sont susceptibles de subir une opération de reconfiguration. Nous identifions ces composants sujets de reconfiguration en utilisant les transitions de la machine à état d'adaptation. Chaque transition de la machine à état définit pour son effet un ensemble d'opérations de reconfiguration pour un ensemble de composants. Cela nous permet d'identifier le type de l'opération de reconfiguration pour chaque composant sujet de reconfiguration. Afin d'assurer que le comportement adaptatif du système n'influencera pas ses caractéristiques temps réels, nous avons à valider le test 5.1 pour chaque composant dans le système. Afin de parcourir que les éléments da la machine à état qui sont sujet potentiel de reconfiguration et effectuer la validation pour ces éléments, nous utilisons l'algorithme 3. Pour ce dernier, nous supposons que chaque composant fournit un service par port.

.

Algorithm 3 validation of real-time application constraints considering adaptive behavior

Require: .
$smAdaptation$, the State Machine of Adaptation
$model$, the system model in request/reply paradigm
Ensure: .
$true$: if the system is validated
$false$: if the system is not valid, and the error source

```
1: for each transition transition ∈ smAdaptation do
2:   for each effect effect ∈ transition do
3:     reconfigOper ← the reconfiguration operation from effect
4:     oldComponent ← the component to be changed from effect
5:     newComponent ← the target component from effect
6:     FDD ← big number
7:     for each provided port port ∈ oldComponent do
8:       for each client component clientComp that use port do
9:         period ← the period used by clientComp to call the service provided by port
10:        newFDD ← period − wcet of the called service
11:        if newFDD < FDD then
12:          FDD = newFDD
13:        end if
14:      end for
15:    end for
16:    if FDD ≥ than wcet of reconfigOper then
17:      log.print(" effect effect is ok ")
18:    else
19:      log.print(" error in the effect effect ")
20:      return false
21:    end if
22:  end for
23: end for
24: return true
```

5.5 Discussion

Après avoir présenté en détail notre approche pour la modélisation et l'analyse du comportement adaptatif des systèmes temps réel, nous la comparons avec les approches les plus similaires présentées dans le chapitre 3. Nous discutons également les apports et les limites de notre approche, cf. tableau 5.3.

- ***Contexte***
  Dans l'approche proposée, nous modélisons dans la partie contexte les éléments de l'environnement pour lesquels le système est susceptible d'avoir un comportement d'adaptation sous la forme des propriétés stéréotypées *Context*. Ces propriétés peuvent être typées par tous les types primitifs d'UML. Cette façon de modélisation nous permet de représenter n'importe quel élément de l'environnement pour pouvoir l'utilisé afin de spécifier le comportement adaptatif. Dans le projet MADAM, le

contexte est modélisé sous la forme des propriétés liées aux composants, cela engendre la redondance de définition des propriétés. Dans notre cas les propriétés sont définies une seule fois et peuvent être utilisées par tous les composants du système.

– ***Variabilité***
Pour la modélisation de la variabilité des implémentations des composants, nous avons utilisé l'héritage [72] et le stéréotype *VariableElement* afin d'indiquer qu'un composant possède et peut utiliser plus qu'une variante en cours d'exécution. Nous ne modélisons pas la variabilité de déploiement et des paramètres des composants. Néanmoins, nous pouvons utiliser notre stéréotype *VariantCondition* pour spécifier le contexte d'utilisation d'une implémentation, d'un déploiement ou d'un certain paramètre de composant, de la même façon qu'un composant possède plusieurs implémentations, il peut posséder également plusieurs déploiements. Ces derniers peuvent être stéréotypés par *VariableElement* afin d'indiquer le contexte d'utilisation de chacun. Dans le projet MADAM, la variabilité est spécifiée aussi à l'aide de la relation d'héritage entre le composant type et ses implémentations. Dans le projet DiVA, toutes les variantes possibles du système sont définies dans un modèle commun, ensuite ces variantes sont composées selon les règles d'adaptation et l'état de l'environnement d'exécution du système. La même idée de base du projet DiVA pour la modélisation de la variabilité est utilisée dans le projet CEA-Frame.

– ***Comportement adaptatif et règle d'adaptation***
Nous modélisons la relation entre le changement du contexte et la sélection de la bonne variante pour chaque composant. Pour la modélisation du comportement nous avons utilisé le comportement modal de MARTE. Nous avons proposé deux automatisations de la création de la machine à état d'adaptation et ses configurations, génération complète et génération partielle. Cette machine à état est créé dans la phase de conception pas en exécution, vu le besoin à la validation. Selon le nombre de variantes dans le système, le nombre des états et des configurations de la machine à état d'adaptation peut être grand. Cependant, ce nombre est réduit à moitié pour chaque contradiction entre deux conditions d'utilisation de deux variantes appartenant à différents composants s'ils ont la même variable d'environnement et avec la même valeur de condition, cf. section 4.4.2. Les trois travaux de MADAM, DiVA et CEA-Frame calculent le modèle du comportement adaptatif en ligne et pas en phase de conception ce qui ne permet pas l'analyse de ce modèle vis-à-vis des caractéristiques temporelles du système. Le travail de Pedro et Bruns ne rentre pas dans le cadre de l'IDM, en plus, les modes sont modélisées manuellement et ne font aucune liaison avec les éléments de l'environnement d'exécution (contexte). Dans le travail d'Etienne Borde, les modes et les configurations associées sont modélisés manuellement et ne font aucune liaison avec les éléments de l'environnement d'exécution (contexte).

– ***Opération de reconfiguration et caractéristiques de temps***
Nous modélisation également les opérations de reconfiguration et leurs caractéristiques temporelles ainsi que les caractéristiques temporelles du système. Ces opérations sont définies dans la classe *Framework*. À notre connaissance aucun travail existant ne traite l'aspect de modélisation des opérations de reconfiguration.

– ***Analyse temporelles du système pour son comportement adaptatif***
Enfin, dans notre approche nous pouvons analyser les caractéristiques temporelles

du système avec la prise en compte de son comportement adaptatif pour deux paradigmes de communication : communication par requête (client/serveur) et communication par flot de données.

Légende : ✓ : supporté ≈ : supporté partiellement – : non supporté **Critères**	**Approches et outils**	Fujaba suite tool [50]	EAST-ADL [40]	MADAM [34]	DiVA [36]	CEA-Frame [39]	Pedro et Burns [56]	Étienne Borde [57]	Notre proposition
Contexte		–	✓	✓	✓	✓	–	–	✓
Variabilité		–	✓	✓	✓	✓	–	–	✓
Comportement adaptatif :		≈	–	✓	✓	✓	✓	✓	✓
• modèle généré		–	–	✓	✓	✓	–	–	✓
• modélisé manuellement		✓	–	–	–	–	✓	✓	✓
• modélisé off line		✓	–	–	–	–	✓	✓	✓
• modélisé on line		–	–	✓	✓	✓	–	–	–
Opérations de reconfiguration		–	–	–	–	–	–	–	✓
Contraintes temporelles du système		✓	✓	–	–	✓	✓	✓	✓
Analyse temporelles du système pour son comportement adaptatif		–	–	–	–	–	–	≈	✓

TABLE 5.3 – Classification des approches selon les critères de la table 2.1

5.6 Conclusion

Dans le chapitre précédent, nous avons présenté une approche de modélisation du comportement adaptatif et la génération du modèle de ce comportement. Dans ce chapitre, nous avons proposé une solution pour pouvoir effectuer une analyse d'ordonnançabilité du système tout en prenant en compte l'influence du comportement adaptatif sur cette analyse. Nous avons utilisé deux sémantiques pour la communication dans les configurations en utilisant deux paradigmes : communication par requête ou flot de données, ce qui détermine le modèle d'exécution. Pour le paradigme de communication par requête nous avons proposé une condition à vérifier pour chaque transition, qui est basée sur la différence de temps entre le WCET et la période de chaque tâche. Cette condition permet de valider le comportement adaptatif de chaque composant dans le système séparément. Autrement dit, elle ne vérifie pas l'influence du comportement adaptatif d'un composant sur un autre composant. Pour le paradigme de flot de données, nous avons proposé une approche basée sur l'utilisation des analyseurs d'ordonnançabilité existants. Dans cette approche, nous générons des *end-to-end flows* qui sont prêts pour une analyse d'ordonnançabilité et qui incluent les opérations de reconfiguration. Ces *end-to-end flows* sont généré à partir d'une activité qui est générée à partir du modèle de système.

Enfin, cette proposition (chapitre 4 et 5) permet de résoudre une lacune dans le domaine du développement des systèmes temps réel adaptable dans le cadre de l'ingénierie dirigée par les modèles. Cette lacune consiste à la modélisation et la gestion du modèle de comportement adaptatif et la validation de celui-ci vis-à-vis des caractéristiques temporelles du système. Dans le chapitre suivant, nous présentons un cas d'étude et une évaluation quantitative de la solution.

CHAPITRE 6

Cas d'étude : robot de cartographie

6.1 Introduction

L'objectif de ce chapitre est de montrer sur un cas d'étude robotique comment modéliser, générer et valider le comportement adaptatif selon l'approche proposée dans les chapitres précédents. D'abord, nous montrons comment utiliser l'approche proposée pour modéliser un exemple robotique à partir d'un cahier des charges afin d'arriver à un modèle prêt pour la génération et la validation du comportement adaptatif. Ensuite, nous illustrons l'utilisation de l'approche de génération du comportement adaptatif, notamment la génération partielle et la génération complète. Nous présentons une comparaison des deux solutions de génération pour deux scénarios de variabilité différents. Cette comparaison est basée sur le nombre des éléments UML généré parmi tous les éléments du comportement adaptatif qui doivent être modélisés pour chacune des solutions. Plus le nombre d'éléments généré est important, plus la solution est meilleure. Nous présentons également la validation de ce comportement adaptatif généré vis-à-vis des caractéristiques temporelles du système pour deux paradigmes de communication, notamment la communication par requête et la communication par flot de données.

6.2 Le robot de cartographie

Dans cette section nous utilisons l'exemple du robot de cartographie présenté dans le projet DiVA [37] afin d'illustrer l'utilisation de notre solution pour la génération du comportement adaptatif (la machine à état d'adaptation) et la validation des caractéristiques temporelles du système vis-à-vis de son comportement adaptatif généré. Nous avons choisi cet exemple pour la nature dynamique de son environnement qui nécessite des changements dynamiques dans le système. L'exemple vise à fournir un système adaptatif pour un robot d'exploration semi-autonome qui construit tout en roulant une carte d'un environnement inconnu.

Le robot est connecté à un système central, qui recueille les données topographiques afin de construire la carte finale et pouvoir fournir des instructions d'orientation au robot. Le robot dispose de trois modes principaux : 1) tourner au ralenti, 2) aller à un endroit précis ou 3) explorer de manière autonome. Il est équipé de trois capteurs différents, qui peuvent être utilisés alternativement pour le routage et pour le dessin de la carte : 1) appareil photo (*Camera*), qui fournit les informations les plus détaillées qui permettent de calculer une carte détaillée, 2) les capteurs infrarouges (*Infrared sensor*), qui peuvent travailler sans sources de lumière et utilisent des ressources limitées, 3) les capteurs à ultrasons (*Ultrasonic sensor*), qui consomment peu de ressources tout en offrant de bonnes capacités de routage.

Tout en étant dessinée par le robot, la carte est stockée soit localement dans la mémoire du robot avec des transmissions périodiques ou directement au système central. Le robot peut employer trois différentes stratégies de routage : 1) une stratégie de routage local qui utilise les capteurs pour naviguer, 2) la stratégie de routage par carte, qui utilise des pré-connaissances du terrain, 3) une stratégie de routage externe qui implique des interactions avec le système central ou un opérateur. En outre, pour construire la carte

lors du déplaçant, le robot peut utiliser soit une stratégie de dessin simple ou détaillée. Selon son environnement, c'est-à-dire, son mode, les conditions de terrain ou les ressources disponibles, le robot doit s'adapter afin d'optimiser la construction de la carte et utiliser les capteurs et les algorithmes appropriés.

6.3 Modélisation du robot de cartographie

6.3.1 Contexte

Selon le cahier des charges du système, le robot doit s'adapter à des éléments de son environnement d'exécution qui sont : le niveau de la lumière, l'état de la mémoire de stockage interne du robot et le mode d'exploration. Pour cela, le modèle de contexte de cet exemple détient quatre propriétés représentant les éléments de l'environnement (cf. figure 6.1) : le niveau de la lumière est représenté par la variable *light*, qui prend trois valeurs : 1 «*Good*» si la lumière est forte, 2 «*Low*» si la lumière est faible et 3 «*None*» si la lumière est inexistante. L'état de la mémoire de stockage interne du robot est représenté par la variable booléenne *lowMemory*, qui prend *true* si la taille de la mémoire libre est faible sinon elle prend *false*. Le mode d'exploration est représenté par la variable *mode*, qui prend 1 «*Idle*» si le robot utilise une stratégie de routage local, 2 «*Explore*» si le robot utilise une stratégie de routage par carte et 3 «*Goto*» si le robot utilise une stratégie de routage externe.

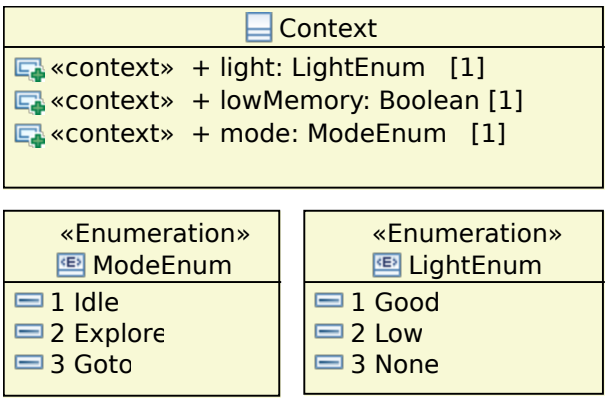


FIGURE 6.1 – Les éléments du contexte

6.3.2 Framework d'adaptation

Afin de s'adapter aux changements de l'environnement, le système utilise des opérations de reconfiguration fournies par le middleware. Ses opérations sont modélisées sur la classe *Framework* tout en précisant leurs caractéristiques temps réel, notamment le pire temps d'exécution de chacune, cf. figure 6.2.

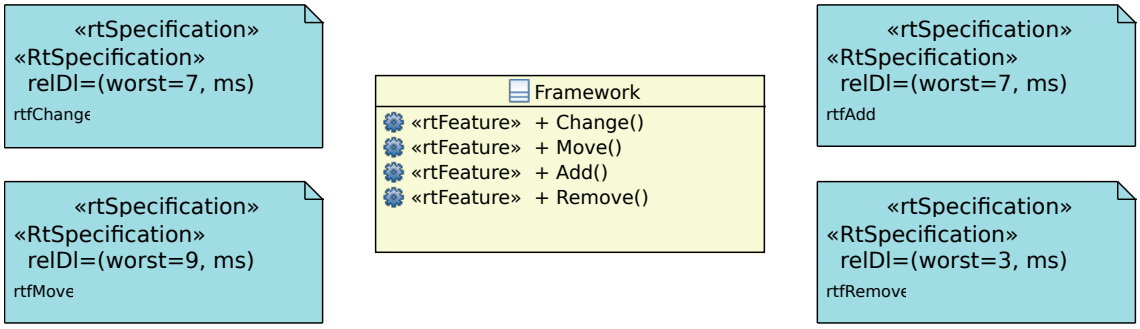


FIGURE 6.2 – Les opérations de reconfiguration

6.3.3 Communication

Nous avons réalisé deux modèles différents pour cet exemple d'application robotique, un modèle pour chacun des deux paradigmes de communication : communication par requête ou flot de données. En ce qui concerne la communication par requête, nous avons défini les interfaces qui seront utilisées pour typer les ports des composants afin de spécifier pour chaque port les services qui seront fournis ou requis. Sur la figure 6.3, nous avons les interfaces et les caractéristiques temps réel (*rtFeature : real time feature*) de leurs opérations.

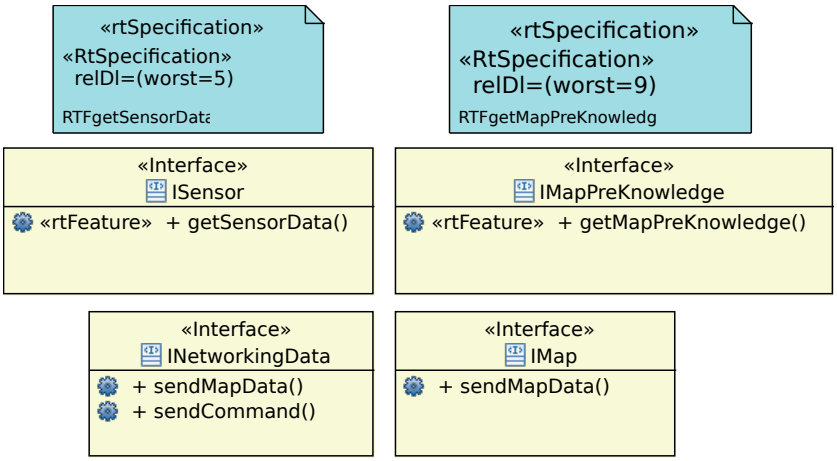


FIGURE 6.3 – Les interfaces de communication entre composants

Quant à la communication par flot de données, nous avons défini les types de données qui seront utilisés pour typer les ports des composants afin de spécifier pour chaque port le type de données qu'il peut faire circuler. Parmi les types de données qui peuvent circuler entre les composants (cf. figure 6.4), nous avons les données venant du capteur (typé avec *SensorData*), les données qui sont envoyés au moteur (typé avec *MotorData*), ... etc.

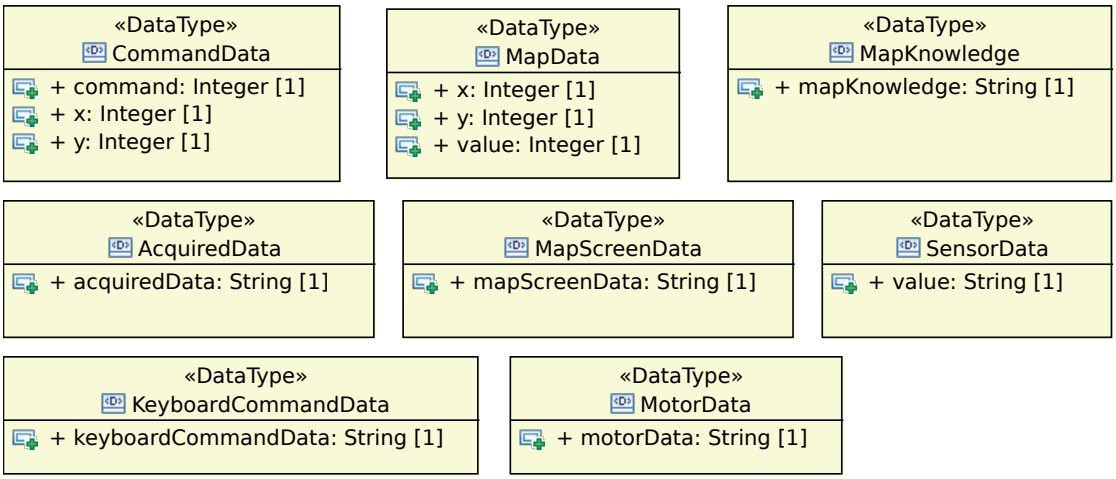


FIGURE 6.4 – Les types de données utilisées pour le flot de données

6.3.4 Les composants

Le système du robot de cartographie est composé de cinq composants : *Router*, *Sensor*, *MapBuilder*, *MapDataTransmitter* et *CentralSystem*. Chacun de ces composants est modélisé par un composant abstrait qui possède au minimum une implémentation. Le composant *Router* implémente les algorithmes qui déplacent le robot, il dispose de trois variantes (implémentations) d'orientation : *LocaleRouting* utilise les informations venant du capteur pour l'orientation, *MapRouting* utilise les informations de la carte locale dans le robot et *ExternalRouting* utilise les informations venant du système central. Le composant *Sensor* (cf. figure 6.5) possède trois variantes, une pour chaque capteur : *Camera*, *Infrared*, *Ultrasonic*.

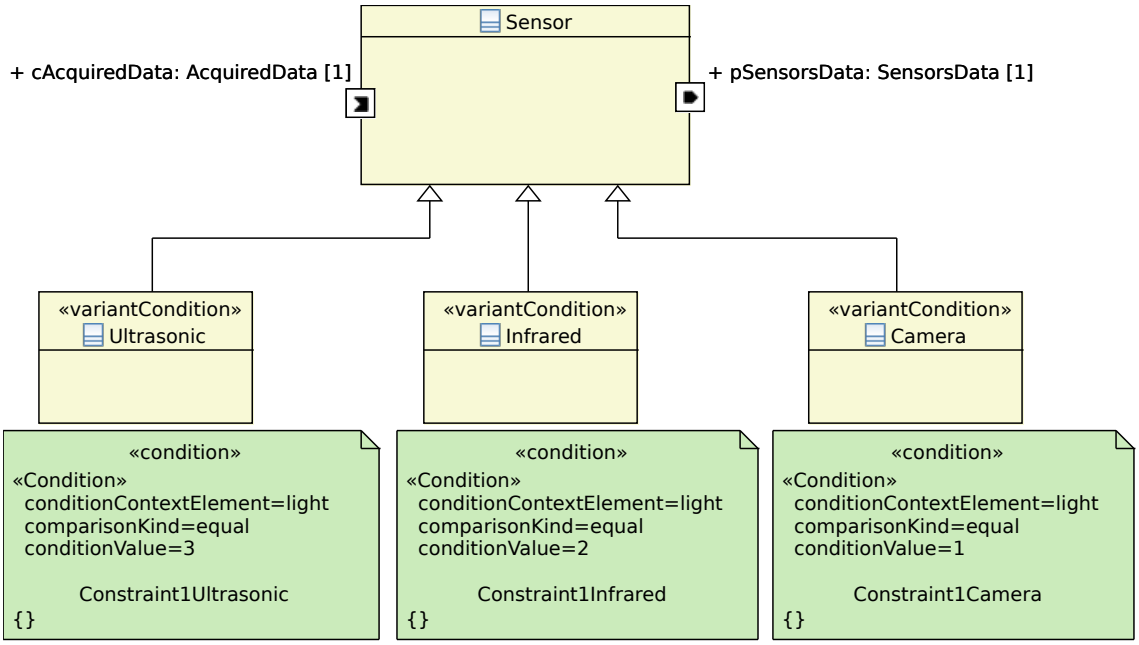


FIGURE 6.5 – Le composant *Sensor* ses variantes et leurs conditions d'utilisation

Les conditions d'utilisation de chaque variante sont spécifiées en liste par le stéréotype *variantConditions* appliqué sur la variante. Chaque condition est détaillée par l'application

du stéréotype condition sur l'élément UML *Constraint*. Par exemple la variante *Camera* sera utilisée si la valeur de la variable d'environnement $light = 3$, (cf. figure 6.5). Le composant *MapBuilder* a deux variantes : *SimpleMap* et *DetailedMap*. Le composant *CentralSystem* dispose d'une seule implémentation.

Dans le modèle à base de communication par flot de données, chaque port est stéréotypé par *FlowPort* de MARTE et le comportement de chaque implémentation d'un composant dans le système est modélisé par une activité, cela nous permet de générer le comportement de chacune des configurations du système sous forme d'une activité, voire dans le cas des composants hiérarchiques. La figure 6.6 présente le diagramme d'activité de la variante *Camera* du composant *Sensor*.

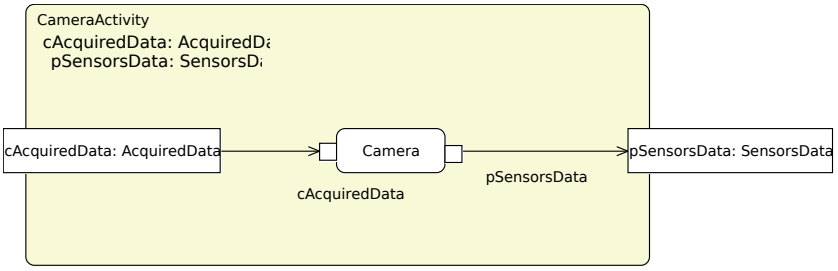


FIGURE 6.6 – Diagramme d'activité de la variante *Camera* du composant *Sensor*

L'implémentation du composant *CentralSystem* est hiérarchique. Elle offre deux fonctionnalités : une pour envoyer les commandes et l'autre pour afficher sur un écran, cf. figure 6.7.

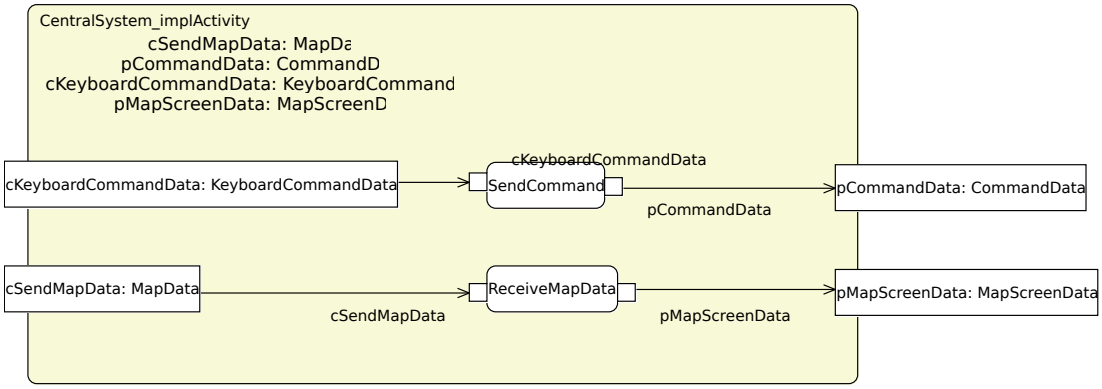


FIGURE 6.7 – Diagramme d'activité de la variante *CentralSystem_impl* du composant *CentralSystem*

6.3.5 Le modèle de base du système

Le modèle du système est composé de cinq propriétés inter-connectées : *router*, *Sensor*, *mapBuilder*, *mapDataTransmitter* et *centralSystem* typées respectivement par les composants : *Router*, *Sensor*, *MapBuilder*, *MapDataTransmitter* et *CentralSystem*, cf. figure 6.8 pour le paradigme de communication requête. À l'exception de la propriété *centralSystem*, les autres propriétés dans le modèle de composition du système sont modélisées comme étant des

éléments variables par l'application du stéréotypé *VariableElement*, car chaque composant de ces propriétés peut utiliser plus d'une implémentation (variante) à l'exécution en fonction de l'état l'environnement.

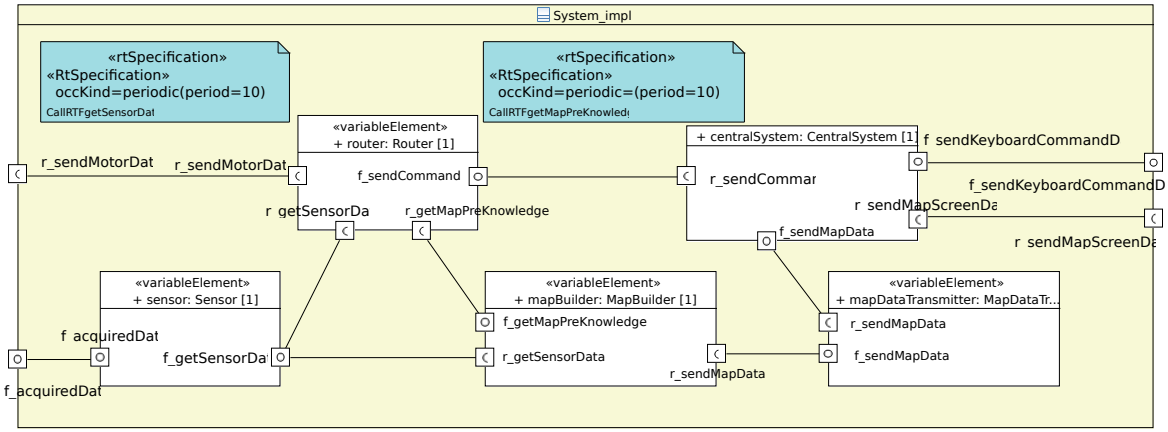


FIGURE 6.8 – Le modèle de composition du système de la communication par requête

Le composant *Router* appelle l'opération *getSensorData* du composant *Sensor* périodiquement chaque $10ms$. Cette contrainte est spécifiée sur le port *r_getSensorData* du composant *Router* à travers l'application du stéréotype *RtFeature* sur le port et le stéréotype *RtSpecification* sur l'élément UML *Constraint*. L'opération *getSensorData* est définie dans l'interface *Isensor* avec $WCET = 5ms$ (cf. figure 6.3). De la même manière, le composant *Router* appelle aussi l'opération *getMapPreKnowledge* du composant *MapBuilder* périodiquement chaque $10ms$. Cette contrainte est spécifiée sur le port *r_getMapPreKnowledge* du composant Router à travers les stéréotypes *RtFeature* et *RtSpecification*. L'opération *getMapPreKnowledge* est définie dans l'interface *IMapPreKnowledge* avec $WCET = 9ms$ (cf. figure 6.3).

La figure 6.9 présente le modèle du système modélisé en utilisant le paradigme de communication de flot de données. Dans ce cas, les contraintes de temps sont spécifiées sur les end-to-end flows qui seront générés dans la phase de validation.

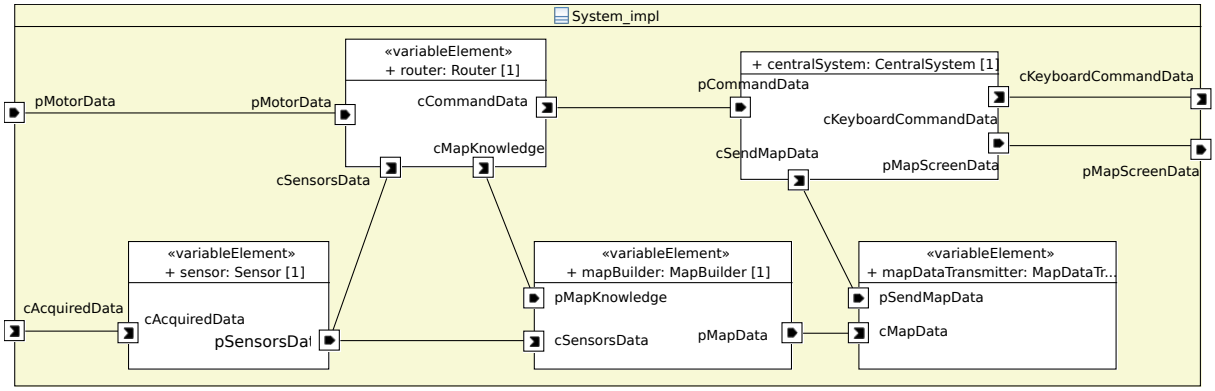


FIGURE 6.9 – Le modèle de composition du système de la communication par flot de donnée

.

Le tableau 6.1 résume le comportement adaptatif du système.

Variants \ Context elements		*light*	*lowMemory*	*mode*
Router	***LocalRouting***	-	-	1 Idle
	MapRouting	-	-	2 Explore
	ExternalRouting	-	-	3 Goto
Sensor	***Camera***	1 Good	-	-
	Infrared	2 Low	-	-
	Ultrasonic	3 None	-	-
MapBuilder	***SimpleMap***	-	True	-
	DetailedMap	-	False	-
MapDataTransmitter	***Streaming***	-	True	-
	Buffering	-	False	-

TABLE 6.1 – Résumé du comportement adaptatif du système

6.4 Génération du comportement adaptatif (machine à état)

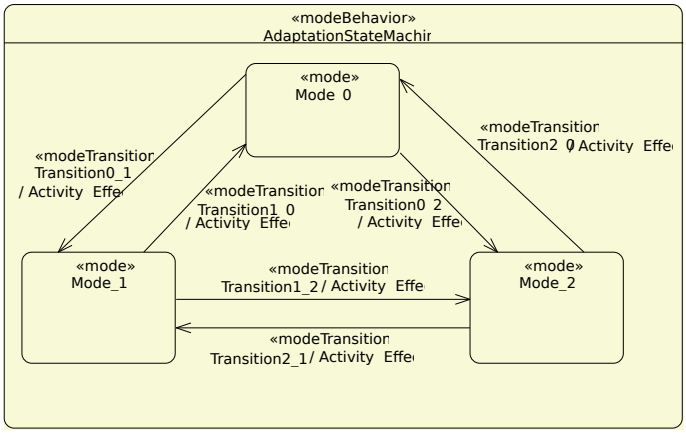
La génération du comportement adaptatif fonctionne indépendamment du paradigme de communication utilisé lors de la modélisation du système. Dans la suite, nous allons utiliser les deux solutions de création du comportement adaptatif (généré et partiellement généré présentées dans le chapitre 4), pour deux scénarios de l'exemple du robot de cartographie : un scénario qui ne contient qu'une seule propriété variable *router* et un deuxième dont toutes les propriétés seront des éléments variables, sauf la propriété *centralSystem*.

6.4.1 Le premier scénario

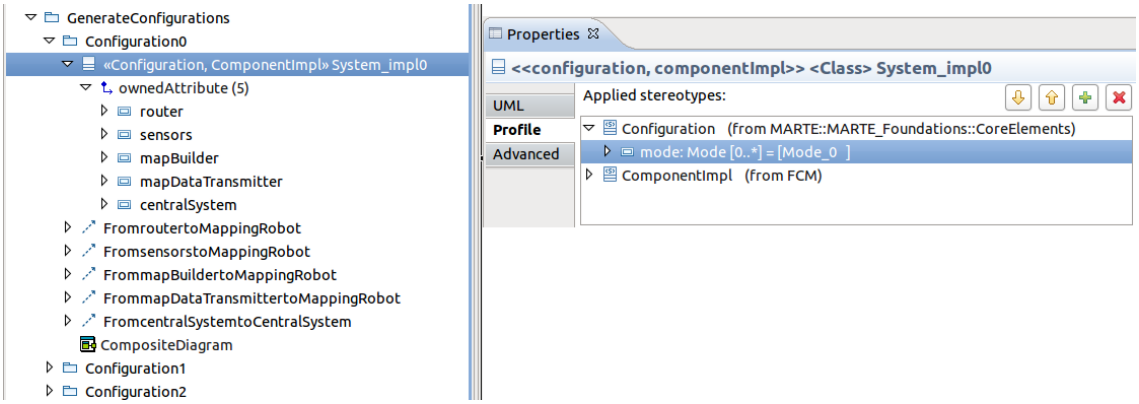
Dans ce scénario nous avons un seul élément variable *router* avec trois variantes *LocalRouting*, *MapRouting* et *ExternalRouting* ce qui donne trois modes possibles avec les configurations respectives.

6.4.1.1 Génération partielle

En utilisant cette solution, le développeur modélise manuellement les modes et les transitions. Au total pour ce scénario, 10 éléments UML seront modélisés manuellement, 1 machine à état, 3 états/modes et 6 transitions. De plus, 45 éléments UML seront générés dont 12 éléments pour les effets des transitions et 3 configurations chacune constituée de 11 éléments UML (cf. figure 6.10).



(a) La machine à état du comportement adaptatif



(b) Les configurations du comportement adaptatif généré

FIGURE 6.10 – Le comportement adaptatif généré

6.4.1.2 Génération complète

En utilisant cette solution, tous les éléments du modèle de comportement adaptatif sont générés, au total 55 éléments UML présentés dans 6.4.1.1.

Sur la figure 6.10a, nous avons une machine à état générée représentant le comportement adaptatif du système composé de 3 modes. Nous avons également, la configuration générée pour chaque mode (cf. figure 6.10b). Le mode associe à une configuration est spécifiée à travers l'attribut *mode* du stéréotype *Configuration* de MARTE (cf. la section 2.4.2).

6.4.2 Le deuxième scénario

Dans ce scénario, nous avons quatre éléments variables. Parmi ces quatre éléments, deux éléments variables possèdent chacun deux variantes, et les deux autres éléments variables chacun d'eux possède trois variantes. Cela donne 36 modes possibles, et pour chaque mode une configuration. Cependant, le système peut utiliser seulement 18 modes, ce nombre est réduit grâce à la dépendance entre les conditions d'utilisation des variantes des composants *MapBuilder* et *MapDataTransmitter* (cf. tableau 6.1). Les variantes *SimpleMap* et *DetailedMap*

du composant *MapBuilder* ne peuvent pas être utilisées respectivement dans le même mode avec les variantes *Buffering* et *streaming* du composant *MapDataTransmitter*. La machine à état du comportement adaptatif possède 612 transitions possibles pour ces 18 modes, cela dans le cas d'un graphe complet. Cependant, dans cet exemple nous n'avons utilisé que les transitions simples, ce qui donne 72 transitions.

6.4.2.1 Génération partielle

En utilisant cette solution, le développeur modélise manuellement les modes et les transitions. Au total pour ce scénario, 91 éléments UML seront modélisés manuellement, 1 machine à état, 18 états/modes et 72 transitions. De plus 342 éléments UML seront générés, dont 144 éléments pour les effets des transitions et 18 configurations chacune constituée de 11 éléments UML.

6.4.2.2 Génération complète

En utilisant cette solution, tous les éléments du modèle de comportement adaptatif sont générés, au total 433 éléments UML. Lorsque le nombre des éléments variables et des variantes augmente, par conséquent, le nombre de modes grandit. Sur cette base, il sera difficile pour le développeur de créer le modèle du comportement adaptatif et de le gérer manuellement, cf. tableau 6.2.

	Méthode de création du comportement adaptatif	***manuel***	***partiellement généré***	***généré***
Scénario un (3 modes)	***Nombre d'éléments créés manuellement***	55	10	0
	Nombre d'éléments générés	/	45	55
Scénario deux (18 modes)	***Nombre d'éléments créés manuellement***	433	91	0
	Nombre d'éléments générés	/	342	433

Scénario un : 3 variantes pour 1 élément variable
Scénario deux : 10 variantes pour 4 éléments variables

TABLE 6.2 – Évaluation du coût développement du comportement adaptatif pour chacune des solutions

6.4.3 Avantages et inconvénients de la génération du comportement adaptatif

Nous pouvons créer le comportement adaptatif manuellement, le généré complètement ou partiellement, chaque méthode possède des avantages et des inconvénients.

– ***Manuellement*** : en utilisant la méthode manuelle, le développeur a plus de liberté pour exprimer directement le comportement adaptatif souhaité pour le système. Cependant, cette façon est coûteuse et source d'erreurs.

- ***Généré*** : en utilisant cette solution, le développement et la modification du comportement adaptatif est plus rapide, surtout dans le cas d'un grand nombre de modes. Toutefois, le développeur exprime indirectement le comportement adaptatif souhaité pour le système.
- ***Partiellement généré*** : le développeur a plus de liberté pour exprimer directement le comportement adaptatif souhaité pour le système. Le développement et la modification du comportement adaptatif est plus rapide dans le cas d'un petit nombre de modes. Cependant, pour un grand nombre de modes, il devient plus lent par rapport à une solution entièrement générée.

6.5 Validation du comportement adaptatif

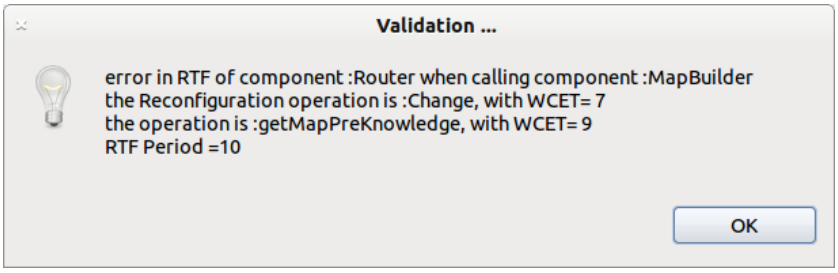
Afin de valider le comportement adaptatif du système vis-à-vis de ses caractéristiques temporelles, nous utilisons le scénario numéro deux de la section précédente dont quatre éléments dans le système sont variables.

6.5.1 Paradigme d'appel d'opération

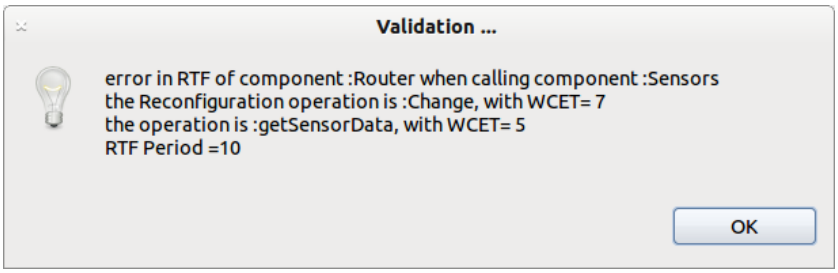
Nous avons utilisé deux cas différents pour les caractéristiques de temps du robot. Dans le premier cas, le $WCET$ de l'opération *getSensorData* est de $5ms$, le $WCET$ de l'opération de reconfiguration *Change* est de $7ms$ et le composant *Router* appel l'opération *getSensorData* avec une période de $10ms$. Dans le deuxième cas, le $WCET$ de *getSensorData* est de $5ms$, le $WCET$ de l'opération de reconfiguration est $7ms$ et la période d'appel est de $20ms$. Pour les deux cas, le FDD [1] est calculé et la condition $FDD \geq ReconfigOperWCET$ est testé pour chaque transition, parmi les 72 transitions, durant laquelle l'un des composants *Sensor* ou *MapBuilder* est changé. D'une manière générale, le test est effectué pour chaque transition dont son effet (*effect*) manipule des composants possèdent des contraintes de temps.

- ***Premier cas*** : l'effet de la transition entre le *Mode_0* et le *Mode_1* (cf. figure 6.10a) remplace la variante *SimpleMap* par *DetailedMap* du composant *MapBuilder*. Dans ce cas : $FDD = p - WCET = 10 - 9 = 1ms$ et $ReconfigOperWCET = 7ms \Rightarrow FDD$ est plus petit que $ReconfigOperWCET$, donc le comportement d'adaptation n'est pas valide et le développeur reçoit un message indiquant cette situation (cf. la figure 6.11a). De la même manière, le comportement n'est pas validé du fait que durant la transition entre le *mode_0* et le *mode_2* la variante *Camera* est remplacée par *Infrared* du même composant *Sensor* et dans ce cas aussi $FDD < ReconfigOperWCET$ (cf. figure 6.11b).
- ***Deuxième cas*** : dans ce cas nous avons augmenté la période ($p = 20$) d'appel pour éviter le problème du scénario précédent. Cette fois-ci pour l'effet de la transition entre le *mode_0* et le *mode_1* : $FDD = p - WCET = 20 - 9 = 11ms$ et $ReconfigOperWCET = 7ms => FDD > ReconfigOperWCET$.

1. Free Duration before the Deadline



(a) Erreur de validation lors du remplacement du composant *MapBuilder*

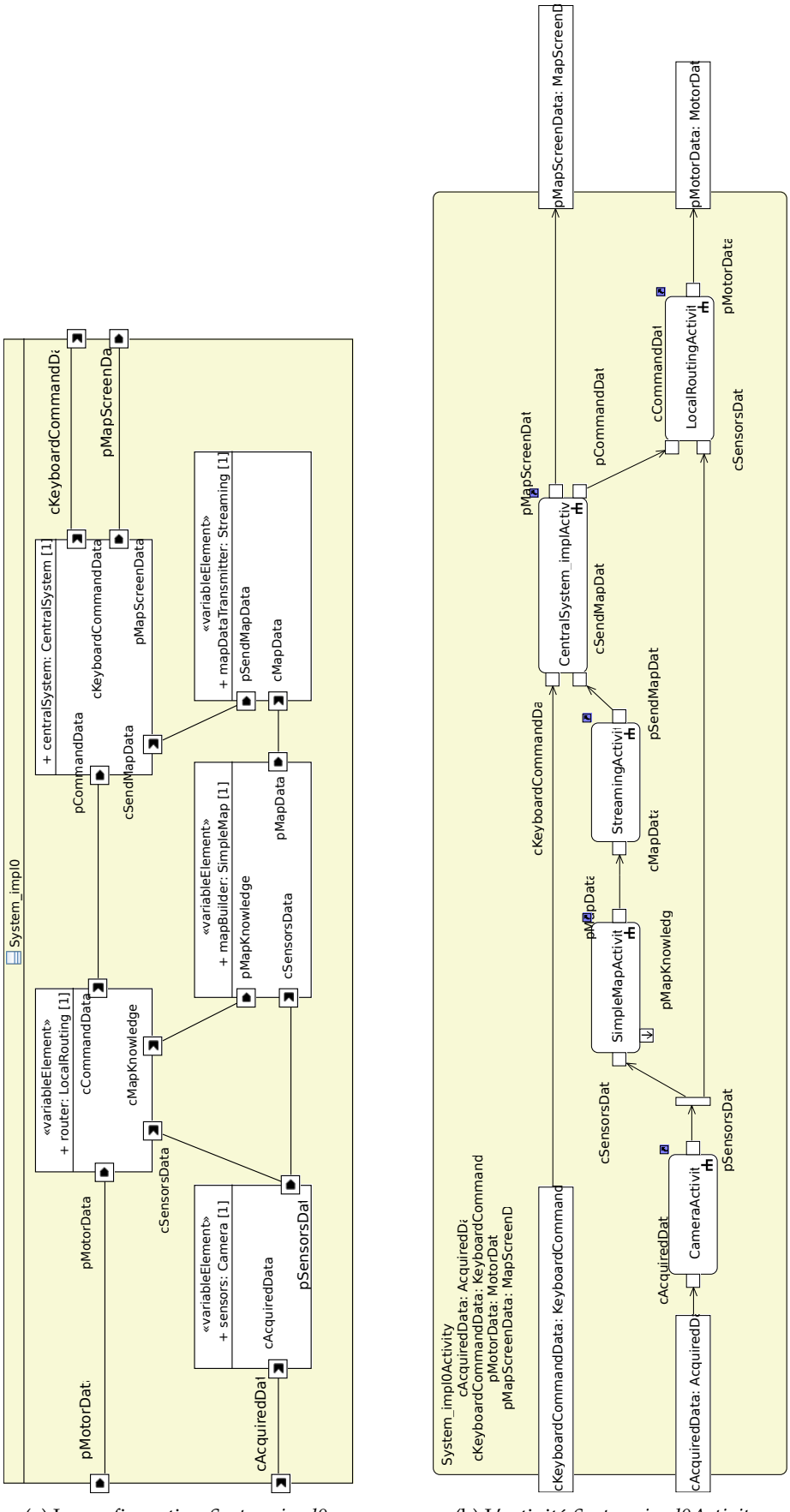


(b) Erreur de validation lors du remplacement du composant *Sensor*

FIGURE 6.11 – Erreurs de validation

6.5.2 Paradigme de flot de données

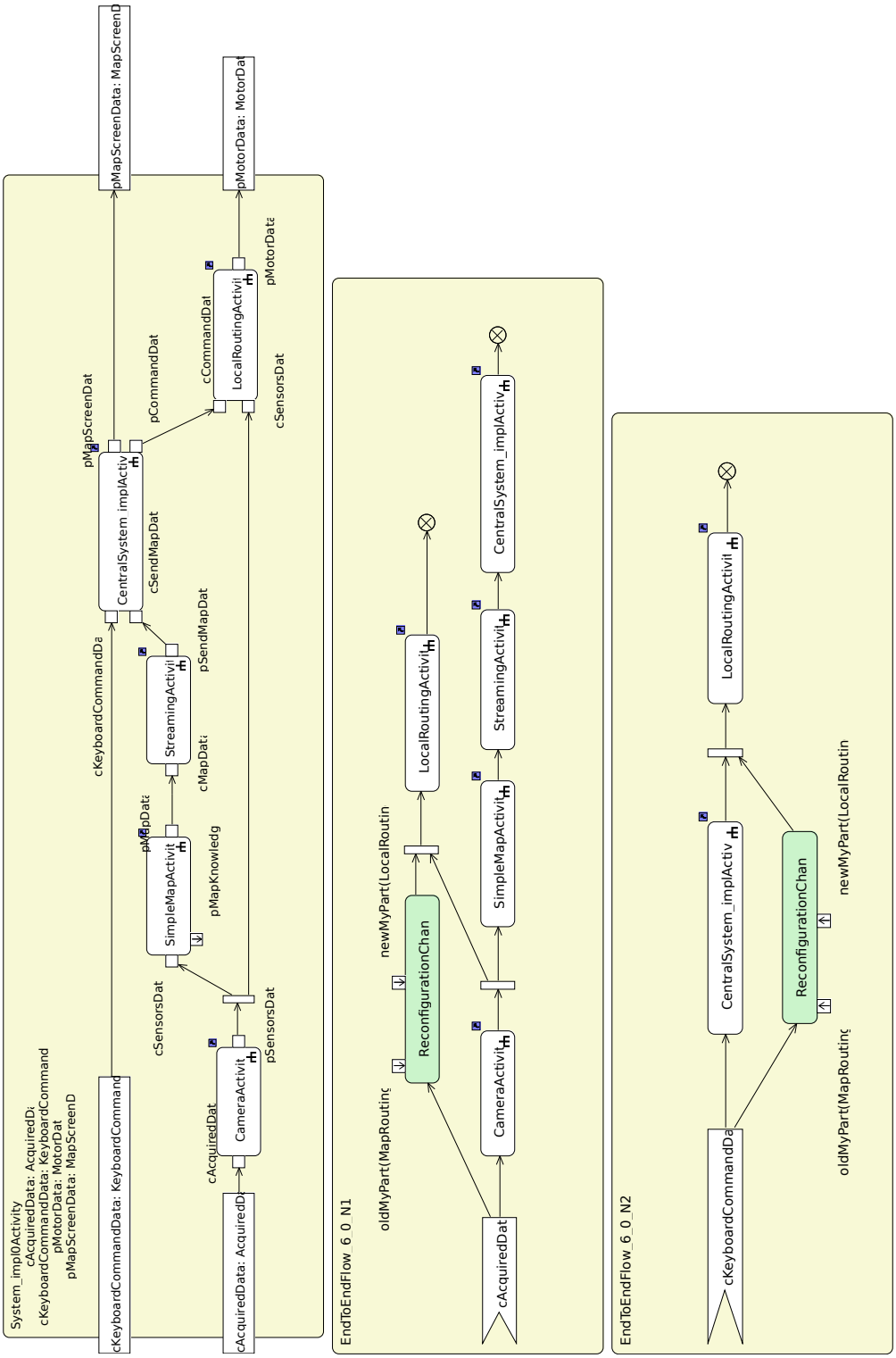
Dans ce paradigme, nous avons utilisé aussi deux scénarios pour les caractéristiques de temps du robot. Les caractéristiques de temps dans ce paradigme sont spécifiées sur les end-to-end flows. Mais avant cela, nous présentons les étapes de génération à partir de la machine à état jusqu'à l'obtention des end-to-end flows. Dans le scénario 2 présenté dans la section 6.4.2, le comportement adaptatif généré comporte 18 modes et une configuration pour chacun. Dans ce qui suit nous utilisons la configuration *System_impl0*, cf. figure 6.12a, du mode *Mode_0* pour l'illustration. Cette configuration est constituée des variantes *LocalRouting*, *Camera*, *SimpleMap* et *Streaming* pour les propriétés *router*, *sensor*, *mapBuilder* et *mapDataTransmitter* respectivement. Dans la figure 6.12b nous avons l'activité généré *System_impl0Activity* pour la configuration *System_impl0*. Chaque port de la classe système *System_impl0* est représenté par un *ActivityParameterNode* dans l'activité. Chaque élément de la classe *System_impl* est mappé à l'activité *System_impl0Activity* en utilisant le tableau 5.1. Ainsi, les activités de toutes les configurations sont générées.



(a) La configuration *System_impl0* (b) L'activité *System_impl0Activity*

FIGURE 6.12 – Le composite de la configuration *System_impl0* et son activité *System_impl0Activity*

Tel que nous avons présenté dans la section 5.3, toutes les transitions de la machine à état sont parcourues et pour chaque activité d'une des configurations cibles de la transition tous les scénarios possibles (end-to-end flow) sont générés. La configuration *System_impl0* possède 4 transitions entrantes. Nous avons déroulé l'exemple pour la transition *Transition6_0* venant de la configuration *System_impl6*. Le point de différence entre les deux configurations *System_impl0* et *System_impl6* est la variante utilisée pour la propriété *router* : la configuration *System_impl6* utilise la variante *MapRouting* par contre la configuration *System_impl0* utilise la variante *LocalRouting*. Le modèle d'activité de la configuration *System_impl0* peut avoir 2 end-to-end flows, cf. figure 6.13. Chaque end-to-end flow est activé par l'un des *Incoming Activity Parameter Node* de l'activité. Les deux end-to-end flows passent par le *Call Behavior Action* : *CentralSystem_implActivity*, mais chacun active une seule sortie vu que l'activité de ce dernier est hiérarchique et composée de 2 flots indépendants, cf. figure 6.7. Une fois les end-to-end flows sont générés, les opérations de reconfiguration définies dans l'effet de la transition sont ajoutées. L'effet de la transition *Transition6_0* comporte une seule opération de reconfiguration qui consiste à remplacer la variante *MapRouting* par la variante *LocaleRouting*, l'appel de l'opération de reconfiguration est ajouté avant la variante sujet de remplacement dans les deux end-to-end flows, cf. figure 6.13. Chaque mode dans la machine à état d'adaptation possède 2 end-to-end flows et 4 transitions entrantes sauf les modes *mode_2* et *mode_4* qui possèdent que 3 transitions entrantes chacun. Ainsi donc le modèle de cet exemple compte 140 end-to-end flows à analyser en vue de valider le comportement adaptatif du robot.



(a) *System_impl0Activity* (b) end-to-end flow 1 (c) end-to-end flow 1

FIGURE 6.13 – Les possibles end-to-end flows de l'activité *System_impl0Activity*

Nos modèles d'end-to-end flow ne sont pas linières, ils contiennent des *forks* et *joins*, ce sont des modèles à événement multiple (*multiple-event model*) [73]. L'analyseur d'ordonnançabilité *MAST* propose dans sa spécification la prise en compte de l'analyse de ces modèles, ainsi que l'analyseur *Optimum*. Cette solution est basée sur la simplification des modèles à événement multiple afin d'avoir des modèles simples, plus de détails dans le papier [73]. Cependant, l'implémentation actuelle de *MAST* (1.4.0.1) et d'optimum n'implémentent pas encore l'analyse des modèles à événement multiple. Selon l'agenda de développement de *MAST* l'analyse des modèles à événement multiple est planifiée pour être prise en compte dans sa prochaine version (1.5.0.0). Compte tenu de cette contrainte, nous n'étions pas en mesure d'analyser ces end-to-end flows.

6.6 Conclusion

Dans ce chapitre, nous avons présenté un cas d'étude sur un robot de cartographie afin de démontrer l'utilisation de l'approche proposée pour la modélisation, la génération et la validation du comportement adaptatif des systèmes temps réel. Cette application robotique est composée de 5 composants logiciels, dont 4 sont des éléments variables où chacun possède plus d'une variante. Nous avons modélisé le système en utilisant les deux paradigmes de communication flot de données et requêtes. Ensuite, nous avons généré le comportement adaptatif pour deux scénarios de variabilité. Le comportement adaptatif est généré en utilisant la génération partielle et la génération complète. La génération permet de faciliter la création du comportement adaptatif quand le nombre de variantes est grand (ce qui implique un grand nombre de configuration). Par contre, l'explosion combinatoire de nombre de configurations est toujours possible si ce n'est pas réduit par les contraintes d'utilisation des variantes. Enfin, la validation de ce comportement est réalisée pour les deux paradigmes de communication : communication par requête et par flot de données.

CHAPITRE 7

Conclusion générale

Pour conclure cette étude, nous présentons dans ce chapitre un bilan des contributions apportées par cette thèse que l'on retrouve de manière détaillée dans les chapitres précédents. Ce bilan est illustré à travers la figure 7.1 qui reprend l'ensemble des étapes de la démarche que nous avons introduit pour la modélisation et l'analyse du comportement adaptatif vis-à-vis des caractéristiques temporelles du système. Nous présentons ensuite les perspectives.

7.1 Rappel de la problématique

Dans les systèmes temps réel à contraintes dures, une analyse d'ordonnançabilité dans la phase de développement est nécessaire. En ce qui concerne les systèmes adaptables, cela nécessite d'avoir un modèle complet du comportement adaptatif. Cependant, la taille du modèle de comportement adaptatif peut être grande, compte tenu de l'explosion du nombre de configurations possible lié au nombre de variantes possible pour chaque composant dans le système. Bien que l'IDM est une approche de développement attractif, néanmoins les efforts nécessaires pour créer ou réviser le modèle du comportement adaptatif sont souvent source d'erreurs et de difficulté, cela variée en fonction de la taille du système. La taille de comportement grandit exponentiellement avec le nombre des éléments variables, si ce n'est pas réduit par des contraintes, cf. section 4.4.2.

Les systèmes non adaptatifs sont fondés principalement sur une configuration unique sur laquelle une analyse d'ordonnançabilité est appliquée. En revanche, les systèmes adaptatifs sont basés principalement sur plusieurs configurations. Chaque configuration est analysée séparément. Cette analyse n'est pas suffisante, vu qu'elle ne prend pas en compte la charge supplémentaire des opérations de reconfigurations. Cette charge supplémentaire influence souvent le fonctionnement du système de point de vue de ses caractéristiques temporelles. L'ingénierie dirigée par les modèles, si elle aide à maîtriser le développement de grands systèmes, n'intègre pas bien le volet de l'analyse temporelle du comportement adaptatif des systèmes temps réel.

7.2 La contribution

Le travail présenté dans cette thèse est focalisé sur la spécification de l'adaptabilité d'un système temps réel et l'étude sur des jeux de configurations prédéfinis de l'impact temporel des actions d'adaptation dynamique.

Pour cela, nous avons présenté une méthodologie outillée basée sur la notion de Mode (extension de machine à état UML) du profil MARTE. Chaque mode représente un comportement possible du système pour un environnement bien déterminé associé à une configuration logicielle. L'application est modélisée au moyen d'une approche par composant (eC3M), ce qui permet le changement des connexions entre composants durant l'exécution. Le changement de configuration est réalisé par des opérations de reconfiguration bien identifiées. L'influence de ces opérations sur le comportement temporel du système est analysée à travers les capacités de prise en charge de ces activités dans l'ordonnancement.

La modélisation du comportement adaptatif dans l'approche développée se base

sur la modélisation du contexte, de la variabilité, des opérations de reconfigurations et de la configuration de base. Le modèle du contexte est la représentation minimale de l'environnement auquel le système doit être adapté. Le contexte est modélisé par une classe UML dont chaque attribut représente un élément de l'environnement. Le modèle de la variabilité représente les alternatives et les variantes possibles des éléments du système. Dans notre approche, les configurations possibles du système sont dérivées des variants des composants, cf. section 4.2. Autrement dit, chaque composant dans le système susceptible d'utiliser plus d'une implémentation en exécution représente un point de variation (élément variable) dans le système. Chaque implémentation représente une variante d'un composant abstrait. Pour chaque variante sont associé des conditions indiquant son contexte d'utilisation. Nous modélisons également les services et opérations fournis par le middleware pour la reconfiguration dynamique. Tous ces services (ex. *change*, *remove*, ... etc.) sont annotés avec MARTE pour spécifier leurs caractéristiques temporelles, notamment le temps d'exécution (C : *computation time*) et le pire temps d'exécution (WCET : *worst case execution time*). Enfin, la structure de base du système est modélisée par l'interconnexion des composants abstraits.

À partir de ces informations, nous pouvons construire la machine à état d'adaptation. Cette machine à état est constituée des modes et de leurs configurations correspondantes. Ainsi que les transitions entre les modes et leurs déclencheurs et effets. Les déclencheurs sont liés aux conditions d'utilisation des variantes. Les effets définissent l'ensemble des appels des opérations de reconfiguration dynamique nécessaires pour le passage de la configuration source à la configuration cible de la transition.

Ensuite, nous avons proposé une solution pour pouvoir effectuer une analyse d'ordonnançabilité du système tout en prenant en compte l'influence du comportement adaptatif sur cette analyse. Nous avons utilisé deux paradigmes pour la communication dans les configurations : communication par requête ou flot de données :

1. Pour le paradigme de communication par requête, nous avons proposé une condition à vérifier pour chaque transition. Cette condition est basée sur la différence de temps entre le WCET et la période de chaque tâche. Cette solution est nécessaire pour la validation de chaque composant séparément, mais elle est non suffisante pour la validation du système complet. Donc une analyse d'ordonnançabilité est nécessaire afin de valider le système.
2. Pour le paradigme de flot de données, nous avons proposé une approche basée sur l'utilisation des analyseurs d'ordonnançabilité existants. Dans cette approche nous générons des *end-to-end flows* incluent les opérations de reconfiguration et prêts pour une analyse d'ordonnançabilité. Ces *end-to-end flows* sont générés à partir d'une activité qui est générée à partir du modèle de système.

Afin de faciliter l'utilisation de l'approche proposée, nous avons implémenté deux solutions de création du comportement adaptatif, qui permettent de générer partiellement ou complètement ce comportement, en vue de faciliter sa création et sa modification. Les deux solutions sont évaluées en termes de nombre des éléments UML créés manuellement ou générés. D'autre part, l'analyse d'ordonnancement est réalisée en utilisant un outil basé sur le formalisme d'UML et le profil MARTE. Cela permet de vérifier que les contraintes temporelles de notre système resteront satisfaites en intégrant les opérations de reconfiguration issues du comportement adaptatif.

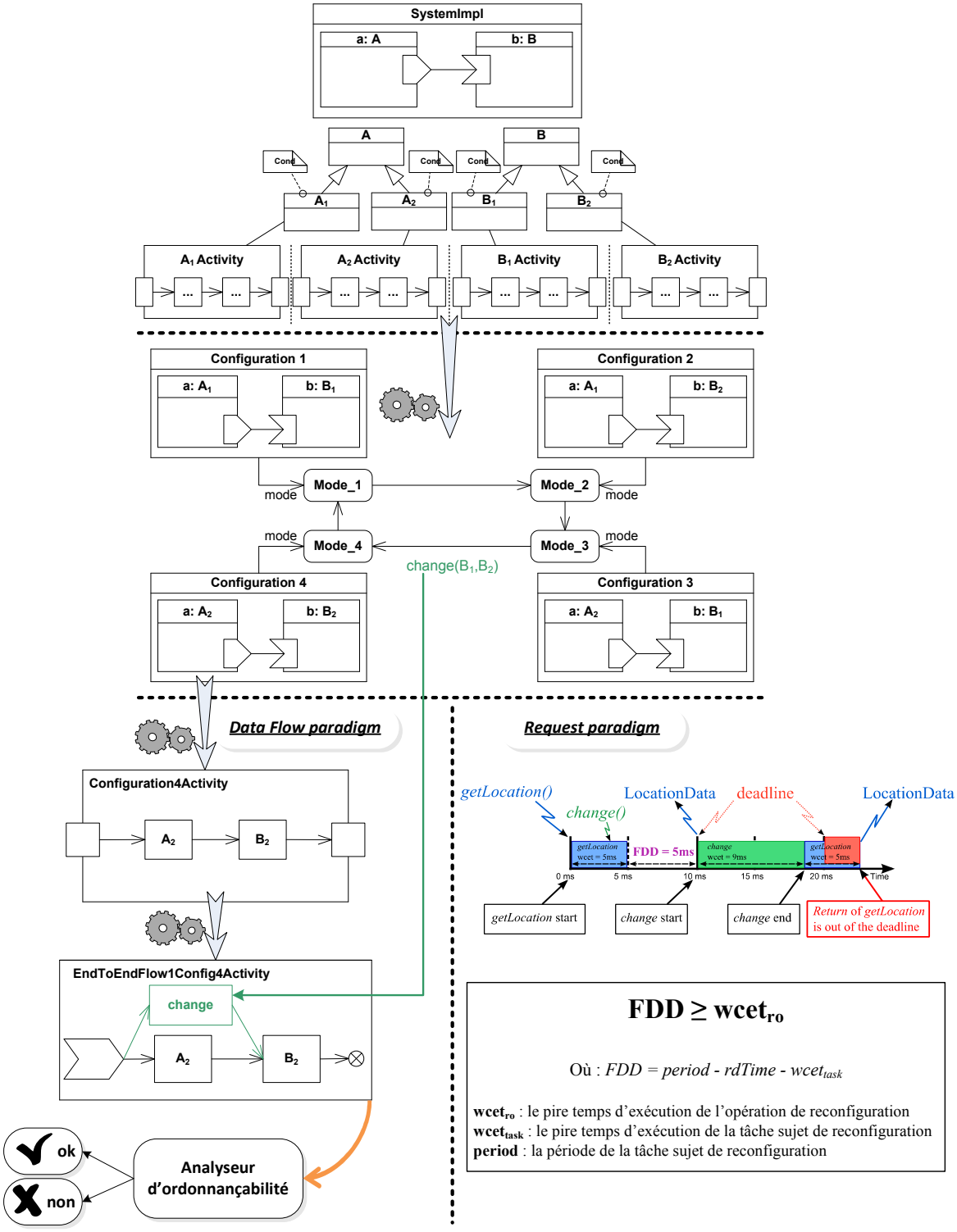


FIGURE 7.1 – Vue globale de la modélisation et l'analyse du comportement adaptatif des systèmes temps réel.

7.3 Perspectives

Nous pensons qu'il est important de poursuivre l'étude de développement des systèmes temps réel adaptable dans le cadre de l'ingénierie dirigée par les modèles. Le travail que nous avons effectué pour la modélisation et l'analyse du comportement adaptatif de ces systèmes est un début. Il serait intéressant de le poursuivre en traitant les points suivants :

7.3.1 À court terme

- Dans cette étude, nous avons pris en compte la variabilité des implémentations des composants au niveau de modélisation et de la génération de la machine à état d'adaptation. Une extension de la solution est souhaitable afin de prendre en compte la variabilité de déploiement des implémentations des composants sur les nœuds.
- Notre contexte de travail est lié au système temps réel aux contraintes dures, cela impose le calcule de toutes les configurations possibles avant l'exécution afin de les validées. Le nombre de configurations varie d'un système à un autre et il peut être très grand. L'explosion combinatoire peut être réduite par l'ajout des contraintes d'utilisation des variantes, en même temps il peut être augmenté par la complexité du système développé. Une recherche pour trouver une solution dans ce cas est souhaitable.
- Pour la machine à état d'adaptation plusieurs analyses liées à la théorie des graphes peuvent être effectuées, notamment la connexité du graphe. Car si le graphe n'est pas connexe cela indique l'existence des configurations non atteignables.
- Étudier la possibilité de grouper des conditions d'utilisation (de la même variante, de la même configuration ou des variantes du même élément variable) par les relations OR et XOR lors de la création du comportement adaptatif.

7.3.2 À long terme

- Il est très intéressant de continuer l'étude de ce travail pour l'implémentation des gestionnaires nécessaires pour l'adaptabilité en cours d'exécution, notamment :
  - ***Gestionnaire de reconfiguration*** : qui implémente des techniques de reconfiguration dynamique pour les types de changement possible dans le système (section 2.2.4). Ce gestionnaire reçoit des ordres de reconfiguration de la part du gestionnaire d'adaptation afin de passer d'une configuration à une autre.
  - ***Gestionnaire de contexte*** : qui s'occupe de l'observation des éléments de l'environnement et met à jour leurs variables correspondantes. Ce gestionnaire envoie également un évènement au gestionnaire d'adaptation si la valeur d'une variable d'environnement traverse la valeur de la condition liée.
  - ***Gestionnaire d'adaptation*** : pour rendre en compte la machine à état en cours d'exécution, afin de savoir réagir aux évènements venant du gestionnaire de contexte et donner les bons ordres au gestionnaire de reconfiguration. Cela nécessite une représentation «*models at runtime*» de la machine à état.
- Dans cette étude nous avons focalisé le travail de validation du comportement adaptatif pour le paradigme de communication par flot de données qui représente le paradigme le plus utiliser par les industriels. Cependant, nous avons proposé

une solution limitée pour le cas de la communication par requête. Il est intéressant d'approfondir et d'aller plus loin pour ce cas de communication.

Table des figures

Bibliographie

[1] Narayanan (Nary) Subramanian and Lawrence Chung. Architecture - Driven Embedded Systems Adaptation for Supporting Vocabulary Evolution. *Principles of Software Evolution, International Symposium on*, 0 :144, 2000.

[2] Nary Subramanian and Lawrence Chung. Software architecture adaptability : an NFR approach. In *Proceedings of the 4th International Workshop on Principles of Software Evolution*, IWPSE '01, pages 52–61, New York, NY, USA, 2001. ACM.

[3] M M Lehman and J F Ramil. Towards a Theory of Software Evolution - And its Practical Impact. *Principles of Software Evolution, International Symposium on*, 0 :2, 2000.

[4] Thomas E. Bihari and Karsten Schwan. Dynamic adaptation of real-time software. *ACM Trans. Comput. Syst.*, 9 :143–174, May 1991.

[5] Juraj Polakovic, Ali Erdem Ozcan, and Jean-Bernard Stefani. Building Reconfigurable Component-Based OS with THINK. In *Proceedings of the 32nd EUROMICRO Conference on Software Engineering and Advanced Applications*, pages 178–185, Washington, DC, USA, 2006. IEEE Computer Society.

[6] Andreas Rasche and Andreas Polze. Redac dynamic reconfiguration of distributed component-based applications with cyclic dependencies. In *Proceedings of the 2008 11th IEEE Symposium on Object Oriented Real-Time Distributed Computing*, pages 322–330, Washington, DC, USA, 2008. IEEE Computer Society.

[7] Andreas Rasche, Marco Puhlmann, and Andreas Polze. Heterogeneous Adaptive Component-Based Applications with Adaptive.Net. In *Proceedings of the Eighth IEEE International Symposium on Object-Oriented Real-Time Distributed Computing*, ISORC '05, pages 418–425, Washington, DC, USA, 2005. IEEE Computer Society.

[8] Craig A. N. Soules, Dilma Da Silva, Marc Auslander, Gregory R. Ganger, and Michal Ostrowski. System Support for Online Reconfiguration. In *In Proc. USENIX Annual Technical Conference*, pages 141–154, 2003.

[9] Jeff Kramer and Jeff Magee. The Evolving Philosophers Problem : Dynamic Change Management. *IEEE Trans. Softw. Eng.*, 16 :1293–1306, November 1990.

[10] M. Vanneschi and L. Veraldi. Dynamicity in distributed applications : issues, problems and the ASSIST approach. *Parallel Comput.*, 33 :822–845, December 2007.

[11] Assia Hachichi. *Container Virtual Machine : Une plate-forme générique pour l'adaptation dynamique des services système dans les intergiciels orientés composants*. PhD thesis, Laboratoire d'Informatique de Paris 6 (lip6), 2006.

[12] Jesper Andersson Department and Jesper Andersson. Issues in Dynamic Software Architectures. In *In Proc. of the Int. Software Architecture Workshop*, pages 111–114. IEEE, 2000.

[13] John A. Stankovic. Misconceptions About Real-Time Computing : A Serious Problem for Next-Generation Systems. *Computer*, 21 :10–19, October 1988.

[14] J. Xu and D. L. Parnas. On Satisfying Timing Constraints in Hard-Real-Time Systems. *IEEE Trans. Softw. Eng.*, 19 :70–84, January 1993.

[15] Pascal Chevochot and Isabelle Puaut. An Approach for Fault-Tolerance in Hard Real-Time Distributed Systems. In *Proceedings of the 18th IEEE Symposium on Reliable Distributed Systems*, SRDS '99, pages 292–, Washington, DC, USA, 1999. IEEE Computer Society.

[16] Pascal Chevochot and Isabelle Puaut. Tolérance aux fautes dans les systèmes répartis temps-réel strict. Techniques et Sciences Informatiques (TSI), 1999.

[17] Emmanuel Grolleau. *Ordonnancement temps réel hors-ligne optimal à l'aide de réseaux de Petri en environnement monoprocesseur et multiprocesseur*. PhD thesis, LISI-ENSMA, 1999.

[18] J. Mc cormick, Frank Singhoff, and J. Hugues. *Building Parallel, Embedded, and Real-Time Applications with Ada*. Cambridge University Press, UK, 365 pages. ISBN-13 : 9780521197168., July 2010.

[19] F. Singhoff, J. Legrand, L. Nana, and L. Marcé. Cheddar : a flexible real time scheduling framework. *Ada Lett.*, XXIV :1–8, November 2004.

[20] http://mast.unican.es/umlmast/.

[21] Gai P., Lipari G., Di Natale M., Serrelli N., and Palopoli L.and Ferrari A. Adding Timing Analysis to Functional Design to Predict Implementation Errors. Detroit, Michigan, April 2007.

[22] Chokri Mraidha, Sara Tucci-Piergiovanni, and Sebastien Gerard. Optimum : a MARTE-based methodology for schedulability analysis at early design stages. *SIGSOFT Softw. Eng. Notes*, 36 :1–8, January 2011.

[23] Cesare Bartolini, Giuseppe Lipari, and Marco Di Natale. From Functional Blocks to the Synthesis of the Architectural Model in Embedded Real-time Applications. In *Proceedings of the 11th IEEE Real Time on Embedded Technology and Applications Symposium*, pages 458–467, Washington, DC, USA, 2005. IEEE Computer Society.

[24] A. L. Davis and R. M. Keller. Data Flow Program Graphs. *Computer*, 15 :26–41, February 1982.

[25] Edward Ashford Lee and David G. Messerschmitt. Static scheduling of synchronous data flow programs for digital signal processing. *IEEE Trans. Comput.*, 36 :24–35, January 1987.

[26] Edward A. Lee and David G. Messerschmitt. Synchronous Data Flow : Describing Signal Processing Algorithm for Parallel Computation. In *COMPCON*, pages 310–315, 1987.

[27] E. A. Lee and T. M. Parks. Dataflow process networks. *Proceedings of the IEEE*, 83(5) :773–801, 1995.

[28] M. Saksena, P. Karvelas, and Y. Wang. Automatic Synthesis of Multi-Tasking Implementations from Real-Time Object-Oriented Models. In *Proceedings of the Third IEEE International Symposium on Object-Oriented Real-Time Distributed Computing*, ISORC '00, pages 360–, Washington, DC, USA, 2000. IEEE Computer Society.

[29] Jean Bézivin. On the unification power of models. *Software and Systems Modeling*, 4 :171–188, 2005. 10.1007/s10270-005-0079-0.

[30] Chokri Mraidha et Benoit Baudry Jean-Marc Jézéquel, Sébastien Gérard. Le génie logiciel et l'IDM : une approche unificatrice par les modèles. In Jean-Marie Favre, Jacky Establier, and Mireille Blay-Fornarino, editors, *L'ingénierie dirigée par les modèles : au-delà du MDA*, chapter 3, pages 53–69. Hermes-Lavoisier, February 2006.

[31] Meta Object Facility (MOF) Core Specification - OMG Available Specification Version 2.0. 2006.

[32] OMG. *OMG Unified Modeling Language (OMG UML), Superstructure, V2.4*. Object Modeling Group, January 2011.

[33] Object Management Group. UML Profile for MARTE : Modeling and Analysis of Real-Time Embedded Systems - version 1.1 - formal/2011-06-022, June 2011.

[34] Jacqueline Floch, Svein Hallsteinsen, Erlend Stav, Frank Eliassen, Ketil Lund, and Eli Gjorven. Using Architecture Models for Runtime Adaptability. *IEEE Softw.*, 23 :62–70, March 2006.

[35] Jan Bosch. *Design and use of software architectures : adopting and evolving a product-line approach*. ACM Press/Addison-Wesley Publishing Co., New York, NY, USA, 2000.

[36] Franck Fleurey, Vegard Dehlen, Nelly Bencomo, Brice Morin, and Jean-Marc Jézéquel. Models in Software Engineering. chapter Modeling and Validating Dynamic Adaptation, pages 97–108. Springer-Verlag, Berlin, Heidelberg, 2009.

[37] Franck Fleurey and Arnor Solberg. A Domain Specific Modeling Language Supporting Specification, Simulation and Execution of Dynamic Adaptive Systems. In *Proceedings of the 12th International Conference on Model Driven Engineering Languages and Systems*, MODELS '09, pages 606–621, Berlin, Heidelberg, 2009. Springer-Verlag.

[38] Pierre alain Muller, Franck Fleurey, and Jean marc Jézéquel. Weaving executability into object-oriented meta-languages. In *in : International Conference on Model Driven Engineering Languages and Systems (MoDELS), LNCS 3713 (2005*, pages 264–278. Springer, 2005.

[39] Sten Lundesgaard, Arnor Solberg, Jon Oldevik, Robert France, Jan Aagedal, and Frank Eliassen. Construction and Execution of Adaptable Applications Using an Aspect-Oriented and Model Driven Approach. In Jadwiga Indulska and Kerry Raymond, editors, *Distributed Applications and Interoperable Systems*, volume 4531 of *Lecture Notes in Computer Science*, pages 76–89. Springer Berlin / Heidelberg, 2007.

[40] Philippe Cuenot, Patrick Frey, Rolf Johansson, Henrik Lönn, Yiannis Papadopoulos, Mark-Oliver Reiser, Anders Sandberg, David Servat, Ramin Tavakoli Kolagari, Martin Törngren, and Matthias Weber. The EAST-ADL architecture description language for automotive embedded software. In *Proceedings of the 2007 International Dagstuhl conference on Model-based engineering of embedded real-time systems*, MBEERTS'07, pages 297–307, Berlin, Heidelberg, 2010. Springer-Verlag.

[41] The ATESST2 Consortium. *EAST-ADL Profile Specification*, June 2010.

[42] M. Jan, C. Jouvray, F. Kordon, A. Kung, J. Lalande, F. Loiret, J. Navas, L. Pautet, J. Pulou, A. Radermacher, and L. Seinturier. Flex-eWare : a flexible model driven solution for designing and implementing embedded distributed systems. Softw : Pract. Exper, 2011.

[43] http://www.flex-eware.org/.

[44] Juan F. Navas, Jean-Philippe Babau, and Olivier Lobry. Minimal yet effective reconfiguration infrastructures in component-based embedded systems. In *Proceedings of the 2009 ESEC/FSE workshop on Software integration and evolution @ runtime*, SINTER '09, pages 41–48, New York, NY, USA, 2009. ACM.

[45] Etienne Borde, Grégory Haïk, and Laurent Pautet. Mode-based reconfiguration of critical software component architectures. In *Proceedings of the Conference on Design, Automation and Test in Europe*, DATE '09, pages 1160–1165, 3001 Leuven, Belgium, Belgium, 2009. European Design and Automation Association.

[46] Douglas C. Schmidt, Aniruddha Gokhale, Balachandran Natarajan, Sandeep Neema, Ted Bapty, Jeff Parsons, Jeff Gray, Andrey Nechypurenko, and Nanbor Wang. CoSMIC : An MDA Generative Tool for Distributed Real-time and Embedded Component Middleware and Applications. 2002.

[47] Aniruddha S. Gokhale, Douglas C. Schmidt, Tao Lu, Balachandran Natarajan, and Nanbor Wang. CoSMIC : An MDA Generative Tool for Distributed Real-time and Embedded Applications. In *International Conference on Distributed Systems Platforms and Open Distributed Processing/Open Distributed Processing*, pages 300–306, 2003.

[48] OMG. CORBA Component Model Specification OMG Available Specification Version 4.0, 2006.

[49] OMG. Deployment and Configuration of Component-based Distributed Applications Specification, 2003.

[50] Sven Burmester, Holger Giese, and Wilhelm Schäfer. Model-Driven Architecture for Hard Real-Time Systems : From Platform Independent Models to Code. In Alan Hartman and David Kreische, editors, *Model Driven Architecture - Foundations and Applications*, volume 3748 of *Lecture Notes in Computer Science*, pages 25–40. Springer Berlin / Heidelberg, 2005.

[51] Sven Burmester, Holger Giese, Martin Hirsch, Daniela Schilling, and Matthias Tichy. The fujaba real-time tool suite : model-driven development of safety-critical, real-time systems. In *Proceedings of the 27th international conference on Software engineering*, ICSE '05, pages 670–671, New York, NY, USA, 2005. ACM.

[52] Sven Burmester, Holger Giese, and Matthias Tichy. Model-Driven Development of Reconfigurable Mechatronic Systems with MECHATRONIC UML. In Uwe Abmann, Mehmet Aksit, and Arend Rensink, editors, *Model Driven Architecture*, volume 3599 of *Lecture Notes in Computer Science*, pages 900–900. Springer Berlin / Heidelberg, 2005.

[53] Holger Giese, Matthias Tichy, Sven Burmester, Wilhelm Schäfer, and Stephan Flake. Towards the compositional verification of real-time UML designs. *SIGSOFT Softw. Eng. Notes*, 28(5) :38–47, September 2003.

[54] Holger Giese and Sven Burmester. Real-Time Statechart Semantics. Technical Report tr-ri-03-239, Lehrstuhl für Softwaretechnik, Universität Paderborn, Paderborn, Germany, 6 2003.

[55] Jorge Real and Alfons Crespo. Mode Change Protocols for Real-Time Systems : A Survey and a New Proposal. *Real-Time Syst.*, 26 :161–197, March 2004.

[56] P. Pedro and A. Burns. Schedulability Analysis for Mode Changes in Flexible Real-Time Systems. *Real-Time Systems, Euromicro Conference on*, 0 :172, 1998.

[57] Etienne Borde. *Configuration et Reconfiguration des Systèmes Temps-Reél Répartis Embarqués Critiques et Adaptatifs*. These, Télécom ParisTech, December 2009.

[58] Serena Fritsch and Siobhán Clarke. TimeAdapt : timely execution of dynamic software reconfigurations. In *Proceedings of the 5th Middleware doctoral symposium*, MDS '08, pages 13–18, New York, NY, USA, 2008. ACM.

[59] Serena Fritsch, Aline Senart, Douglas C. Schmidt, and Siobhán Clarke. Time-bounded adaptation for automotive system software. In *Proceedings of the 30th international conference on Software engineering*, ICSE '08, pages 571–580, New York, NY, USA, 2008. ACM.

[60] Ji Zhang, Heather J. Goldsby, and Betty H.C. Cheng. Modular verification of dynamically adaptive systems. In *Proceedings of the 8th ACM international conference on Aspect-oriented software development*, AOSD '09, pages 161–172, New York, NY, USA, 2009. ACM.

[61] Fabiano Cruz, Raimundo Barreto, Lucas Cordeiro, and Paulo Maciel. ezRealtime : a domain-specific modeling tool for embedded hard real-time software synthesis. In *Proceedings of the conference on Design, automation and test in Europe*, DATE '08, pages 1510–1515, New York, NY, USA, 2008. ACM.

[62] Sébastien Gérard, François Terrier, and Yann Tanguy. Using the Model Paradigm for Real-Time Systems Development : ACCORD/UML. In *in OOIS'02-MDSD. 2002*, pages 260–269. Springer, 2002.

[63] Sébastien Gérard. *Modélisation UML exécutable pour les systèmes embarqués de l'automobile.* PhD thesis, Université d'Evry, October 2000.

[64] http://www-list.cea.fr/index.htm.

[65] Ansgar Radermacher, Arnaud Cuccuru, Sebastien Gerard, and François Terrier. Generating execution infrastructures for component-oriented specifications with a model driven toolchain : a case study for MARTE's GCM and real-time annotations. *SIGPLAN Not.*, 45 :127–136, October 2009.

[66] Andreas Rasche and Andreas Polze. Configuration and Dynamic Reconfiguration of Component-Based Applications with Microsoft .NET. In *Proceedings of the Sixth IEEE International Symposium on Object-Oriented Real-Time Distributed Computing*, ISORC '03, pages 164–, Washington, DC, USA, 2003. IEEE Computer Society.

[67] Manas Saksena and Panagiota Karvelas. Designing for schedulability : integrating schedulability analysis with object-oriented design. In *Proceedings of the 12th Euromicro conference on Real-time systems*, Euromicro-RTS'00, pages 101–108, Washington, DC, USA, 2000. IEEE Computer Society.

[68] Zonghua Gu and Zhimin He. Real-Time scheduling techniques for implementation synthesis from component-based software models. In *Proceedings of the 8th international conference on Component-Based Software Engineering*, CBSE'05, pages 235–250, Berlin, Heidelberg, 2005. Springer-Verlag.

[69] Jamison Masse, Saehwa Kim, and Seongsoo Hong. Tool Set Implementation for Scenario-based Multithreading of UML-RT Models and Experimental Validation. In

Proceedings of the The 9th IEEE Real-Time and Embedded Technology and Applications Symposium, RTAS '03, pages 70–, Washington, DC, USA, 2003. IEEE Computer Society.

[70] Anders Wall, Magnus Larsson, and Christer Norström. Towards an Impact Analysis for Component Based Real-Time Product Line Architectures. In *EUROMICRO*, pages 81–89. IEEE Computer Society, 2002.

[71] Paweł Rodziewicz. *Timing and scheduling analysis of real-time object-oriented models*. PhD thesis, Concordia University, 1998.

[72] Hassan Gomaa and Diana L. Webber. Modeling Adaptive and Evolvable Software Product Lines Using the Variation Point Model. In *Proceedings of the Proceedings of the 37th Annual Hawaii International Conference on System Sciences (HICSS'04) - Track 9 - Volume 9*, HICSS '04, pages 90268.3–, Washington, DC, USA, 2004. IEEE Computer Society.

[73] J. J. Gutiérrez García, J. C. Palencia Gutiérrez, and M. González Harbour. Schedulability analysis of distributed hard real-time systems with multiple-event synchronization. In *Proceedings of the 12th Euromicro conference on Real-time systems*, Euromicro-RTS'00, pages 15–24, Washington, DC, USA, 2000. IEEE Computer Society.