Sommaire

Liste des figures

Liste des Tableaux

Introduction

Les systèmes embarqués envahissent de plus en plus notre vie quotidienne. D'une grande diversité, ces systèmes occupent une place prépondérante dans plusieurs domaines de l'industrie comme la télécommunication, l'automobile, l'avionique ou encore l'aérospatial. Ces systèmes sont composés d'une partie matérielle et d'une autre logicielle et possèdent la particularité d'être enfouis dans un milieu avec lequel ils sont en interaction permanente. Cette interaction avec le milieu s'effectue le plus souvent en temps réel. Les systèmes embarqués doivent donc être des systèmes temps réel [REAL03].

La conception de ces systèmes devient de plus en plus complexe, du fait de l'hétérogénéité grandissante des applications (traitement de signal multimédia, communication, sécurité) ainsi que les architectures de déploiement (processeurs multicores et architectures dédies, systèmes sur puce (SoCs)). Face à cette complexité, ces systèmes se trouvent alors difficiles à spécifier et les possibilités d'erreurs sont de plus en plus nombreuses. Leur disfonctionnement, à cause de la surcharge ou de la terminaison de quelques services après un délai imposé, peut avoir de graves conséquences (économiques, judiciaires, humaines, etc.). De ce fait, plusieurs contraintes s'imposent lors de la conception de ces systèmes, essentiellement celles temps réels. La vérification des propriétés du système pour une étape préliminaire pourrait diminuer la dimension du problème.

Actuellement, la modélisation orientée objets soutenue par le standard UML (Unified Modeling Language) apporte des solutions efficaces à de tels problèmes, ceci est réalisé via l'extension et/ou la restriction de ce langage par l'intermédiaire de profil [UML04]. Cependant, les capacités de spécification du comportement temps réel d'une application ne sont pas encore complètement satisfaisantes. En effet, ces méthodes, récemment industrialisées, fournissent des solutions en terme de spécification de la concurrence d'une application, mais restent insuffisantes notamment, pour l'expression des contraintes non fonctionnelles d'une application et pour l'intégration de la modélisation de l'RTOS (Real Time Operating System) [VIV02] lors de la conception du système.

Ainsi, pour remédier à ces problèmes, nous essayons dans ce mémoire de Mastère de modéliser un RTOS tout en définissant les sémantiques temporelles [ARN07]. Pour cela, nous définissons un nouveau modèle relatif à un RTOS. Ce modèle tient compte explicitement des instructions conditionnelles présentes dans le code des tâches et permet ainsi de prendre en compte l'ensemble des durées d'exécution des tâches et le comportement réel de l'application vis à vis de la gestion des ressources. Nous sommes alors amenés à définir le problème de l'ordonnancement [YVO05] en définissant les variations sémantiques aux statecharts associées à l'état d'un processus. Enfin, après l'implantation des statecharts et à l'aide de l'ingénierie dirigée par les modèles nous montrons comment nous pouvons générer automatiquement le code relatif à un RTOS.

Pour ce faire, nous avons divisé ce document en cinq chapitres :

Dans le premier chapitre, nous rappellerons les principes fondamentaux liés à notre travail. Nous présenterons tout d'abord un aperçu sur la notion de système temps réel et en particulier son architecture. Nous détaillerons ensuite les caractéristiques de l'exécutif temps réel.

Dans le deuxième chapitre, nous effectuerons une étude des travaux menés dans divers projets liés au domaine de conception des systèmes temps réel. Nous étudierons plus en détail la notion de profil UML, et nous nous attacherons à synthétiser les caractéristiques de quelques profils.

Dans le troisième chapitre, nous présenterons les concepts liés à l'ingénierie dirigée par les modèles (IDM) tout en insistant sur l'approche MDA (Model Driven Architecture). De plus, une attention est mise sur l'architecture, les principes et les standards de cette approche en mettant l'accent sur la transformation des modèles

Le quatrième chapitre expose notre solution conceptuelle pour la mise en oeuvre de la modélisation de l'RTOS lors de la modélisation d'un système temps réel, notamment en adoptant la démarche dirigée par les modèles. Nous expliquerons les choix qui nous ont menés au modèle de l'RTOS, et nous détaillerons chacune de ses composantes.

Le dernier chapitre constitue une partie plus applicative, puisqu'il consiste à présenter les expérimentations que nous avons menées en parallèle à notre étude. Nous montrerons les différentes solutions techniques que nous avons exploité pour implanter les statecharts et générer automatiquement le code.

En conclusion générale, nous résumons la démarche qui nous a amené à la modélisation du processus de développement dirigé par les modèles pour la spécification et la mise en oeuvre de l'exécutif temps réel. Nous terminons enfin en exposant les perspectives liées à ce travail, tant du point de vue de la modélisation que de la réalisation.

C H A P I T R E

1

Systèmes temps réel

1. Introduction

L'objectif de ce chapitre est de présenter le contexte scientifique dans lequel est réalisée ce travail de Mastère. Les notions mises en avant sont celles qui ont été retenues pour la réalisation de notre étude, et ne constituent pas une étude exhaustive. Cette présentation du contexte scientifique s'articule autour du domaine des systèmes temps réel, couvrant les aspects architecturaux, techniques et conceptuels sur lesquels sont bâtis notre travail.

Nous présentons de manière générale les systèmes temps réels. Nous rappelons leurs architectures et nous focalisons sur les caractéristiques d'un système d'exploitation temps réel. Nous en synthétisons ensuite les principales fonctionnalités.

2. Définition d'un Système Temps Réel

Il existe de nombreuses définitions des Systèmes Temps Réel. Une première définition tirée de [STA88], décrit un système Temps Réel comme: «tout système informatique dont le bon fonctionnement ne dépend pas uniquement de la correction algorithmique et logique mais également des dates d'arrivée des résultats». Contrairement à la notion de correction temporelle qui est bien mise en évidence, le caractère réactif est loin d'être explicitement défini. Une deuxième approche qualifie les systèmes Temps Réel comme étant : «des systèmes ouverts répondant constamment aux sollicitations de leur environnement en produisant des actions sur celui-ci». Cette définition insiste sur la notion de servitude vis à vis du procédé contrôlé mais sans évoquer l'aspect temporel. C'est au CNRS [CNR88] que nous pouvons enfin trouver la définition suivante qui concilie entre les deux paradigmes primordiaux des Systèmes Temps Réel : «Peut être qualifiée de temps réel toute application mettant en oeuvre un système informatique dont le fonctionnement est assujetti à l'évolution dynamique de l'état d'un environnement (procédé) qui lui est connecté et dont il doit contrôler le comportement».

Dans le but d'affiner cette définition, par le biais d'introduction des notions de critères temporels, nous pouvons nous référer à la définition donnée par [ALA92] :«Une application temps réel constitue un système de traitement de l'information ayant pour but de commander un environnement imposé en respectant les contraintes de temps et de débit (temps de réponse à un stimulus, taux de perte d'information toléré par entrée) qui sont imposées à ses interfaces avec cet environnement».

Ainsi, les systèmes temps réel [REAL03] se distingue par leur capacité permettant aux applications qu'ils gèrent (nommées : applications temps réel) de réagir à des événements et/ou d'atteindre des résultats selon des contraintes de temps fixées antérieurement. Ils sont notamment susceptibles de disposer de mécanisme de contrôle des automates, des robots, des chaînes de production, des véhicules, des centrales nucléaires…

3. Applications Temps réel

Les définitions présentées précédemment des Systèmes Temps Réel ont mis l'accent sur deux éléments distincts : une ou plusieurs entités physiques constituant le procédé, dont le rôle est d'agir et de détecter, et un contrôle informatique, nommé contrôleur ou application temps réel qui est le décideur des actions (ou réactions) du procédé. Le contrôleur reçoit des informations sur le milieu du procédé à l'aide de capteurs et commande les changements d'état du procédé à travers des actionneurs.

3.1. Applications concurrentes

La dynamique des périphériques (ou interfaces) du procédé et de son environnement détermine celle de l'interaction entre le procédé et le contrôleur. De la même façon que l'environnement connaît des transformations en parallèle avec les périphériques du système, le contrôleur, c'est à dire l'application temps réel, est censé de refléter ce parallélisme.

Pour gérer toutes ces entités interagissantes, il est donc indispensable de développer des techniques logicielles capables de traiter les informations reçues des capteurs sur l'unité de calcul pour fournir les actions appropriées.

3.2. Notion de Tâche

La notion de capteurs et d'actionneurs introduit implicitement, en terme logiciel, l'utilisation de différentes tâches permettant de les piloter. Ce sont des programmes séquentiels dédiés au traitement d'un des composants du système Temps Réel. A titre d'exemple, un programme Temps Réel peut être constitué d'un ensemble de tâches tels que :

- des exécutions périodiques de mesures de différentes grandeurs physiques (pression, température, accélération, etc.). Ces valeurs peuvent être comparées à des valeurs de consignes liées au cahier des charges du procédé

- des traitements à intervalles réguliers ou programmés
- des traitements en réaction à des évènements internes ou externes : dans ce cas les tâches doivent être aptes à accepter et à analyser en accord avec la dynamique du système, les requêtes liées à ces évènements. Nous considérons ainsi, une Application Temps Réel comme étant une application multitâches.

3.3. Interaction entre les tâches

Les tâches, dont les comportements sont séquentiels, peuvent interagir entre elles pour garantir le bon fonctionnement global de l'application que le système commande. Il est donc nécessaire de fournir parallèlement à ces tâches des moyens de communication et de synchronisation susceptibles de gérer tous les problèmes liés aux accès à des ressources communes comme par exemple les périphériques (mémoires, imprimantes, etc.), ou l'exécution des tâches ordonnées par des critères de précédence.

Dans le cas du partage de ressources, certaines d'entre elles peuvent être bornées en nombre d'accès simultanés. Dans ce cas, nous évoquons la notion de ressources critiques. Pour assurer un bon fonctionnement de l'application, il est nécessaire de mettre l'accès à ces ressources en exclusion mutuelle. Il faut s'assurer qu'il y ait bien au plus le nombre maximum autorisé de tâches simultanément en section critique, c'est à dire qui utilisent simultanément la ressource. De plus, il convient de garantir la non préemption des ressources en cours d'utilisation. L'accès à ces ressources peut de plus s'effectuer en mode lecture ou écriture, chacun possédant son propre nombre d'accès simultanés autorisés.

Les critères de précédence des tâches sont souvent issus soit d'un désir d'échange de données entre deux tâches, soit de la volonté de synchroniser deux tâches pour que la suite de leur exécution se fait en parallèle par un mécanisme de Rendez-Vous. Dans le premier cas, nous parlons d'une tâche émettrice et d'une tâche réceptrice.

Nous identifions le concept de tâches indépendantes lorsque l'application n'utilise ni ressources critiques, ni synchronisation.

4. Architecture des Systèmes Temps Réel

Un système temps réel est formé de deux composantes : une matérielle et une autre logicielle.

Dans cette partie, on ne parle pas de la cible architecturale à générer à un niveau d'abstraction dans le cadre de la conception des circuits intégrés (CI) ou encore dans le cadre du co design. On se place plutôt dans le cadre du génie logiciel en parlant d'une architecture matérielle qui est à base de processeurs et d'une architecture logicielle mettant l'accent sur la structure et l'agencement des composantes logicielles qui forment l'application à concevoir.

4.1. Architecture matérielle

Les systèmes temps réel peuvent être classés selon leur couplage avec des éléments matériels avec lesquels ils sont en interaction. Ainsi, l'application concurrente et le système d'exploitation qui lui est associé peuvent se trouver :

- soit directement dans le procédé contrôlé : dans ce cas, il s'agit des systèmes embarqués. Le procédé est pour la plupart très spécialisé et fortement dépendant du calculateur. Les exemples de systèmes embarqués sont nombreux : contrôle d'injection automobile, stabilisation d'avion, électroménager... C'est le domaine des systèmes spécifiques intégrant des logiciels sécurisés optimisés en encombrement et en temps de réponse.
- soit le calculateur est détaché du procédé : c'est souvent le cas lorsque le procédé ne peut être physiquement couplé avec le système ou dans le cas général des contrôle/commandes de processus industriels. Dans ce cas, les applications utilisent généralement des calculateurs industriels munis de systèmes d'exploitation standards ou des automates programmables industriels comme dans les chaînes de montage industrielles par exemple.

En intégrant la notion de calculateur ou de processeur, nous distinguons trois grandes catégories d'architecture matérielle pour les Systèmes Temps Réel en fonction de leur richesse en terme de nombre de cartes d'entrée/sortie, de mémoires, de processeurs et de la présence de réseaux.

- L'architecture monoprocesseur : un unique processeur exécute toutes les tâches de l'application concurrente. Dans ce cas, la notion de parallélisme n'a plus vraiment de sens puisque le temps processeur est partagé entre toutes les tâches. Nous parlons plutôt de pseudo-parallélisme ou d'entrelacement des exécutions. En effet, le parallélisme des tâches semble réel à l'échelle de l'utilisateur mais le traitement sur l'unique processeur s'opère de façon séquentielle.

- L'architecture multiprocesseurs : l'exécution de toutes les tâches est ici répartie sur n processeurs partageant une unique mémoire centrale. La coopération entre tâches se fait par partage des informations placées en mémoire. Le traitement est donc ici réellement parallélisé.
- L'architecture distribuée : c'est le cas des architectures multiprocesseurs ne partageant pas de mémoire centrale. Ces processeurs sont reliés entre eux par l'intermédiaire de réseaux permettant d'assurer les communications entre les différentes tâches. Une ferme d'ordinateurs est un exemple typique de cette architecture. La coopération se fait ici par communication par réseau.

Par la suite nous nous placerons dans le cas des architectures monoprocesseur dans toutes les parties de notre travail.

4.2. Architecture logicielle

L'architecture logicielle d'un système Temps réel est divisée en deux couches. La première consiste en une application concurrente composée d'un ensemble de tâches. Nous utilisons également le terme d'applications multitâches. La deuxième, de plus bas niveau, joue le rôle d'un système d'exploitation minimal chargé de faire le lien entre le procédé physique et l'application multitâches.

Ce système d'exploitation, appelé exécutif Temps Réel, de par la considération de l'asynchronisme, est dirigé par les évènements, ceux-ci pouvant provenir de différentes sources :

- du procédé physique par l'intermédiaire d'interruptions matérielles associées à chaque évènement.
- du temps : chaque système est muni d'une horloge Temps Réel pouvant générer des interruptions.
- de l'application multitâche lorsque par exemple l'exécution d'une tâche est conditionnée par l'exécution d'autres tâches. Dans ce cas il faut que l'exécutif retarde l'exécution de cette tâche pour permettre au préalable au processeur d'exécuter les autres.

L'exécutif temps réel propose différents services et garanties facilitant l'exécution et la communication des tâches. Ces services appelés primitives temps réel peuvent être directement utilisés dans les tâches et sont de différentes natures :

- **Gestion des tâches** : Celles ci changent d'état au cours de leur utilisation dans le système. Elles sont toutes initialement inexistantes dans le système. Elles sont alors "créées" puis réveillées ce qui les positionnent dans l'état "prête". Un mécanisme logiciel de choix décide alors d'élire une tâche parmi celles dans l'état "prête" pour que le processeur la traite. Dans ce cas l'état de la tâche passe à "exécutée". De cet état, une tâche peut soit être préemptée par une autre tâche, dans ce cas elle retourne dans l'état "prêt", soit être bloquée par une synchronisation, ce qui la fait passer à l'état "attente", soit enfin elle termine son exécution et passe dans l'état "terminée" avant de disparaître du système
- **Gestion des ressources partagées** : Nous avons vu que certaines ressources peuvent être critiques et qu'elles doivent alors être utilisées en exclusion mutuelle. L'utilisation de ces ressources nécessite des techniques permettant de garantir le respect de l'exclusion mutuelle. Par exemple, la plus simple consiste à masquer les interruptions durant l'utilisation des ressources, ce qui empêche l'exécutif temps réel de traiter les nouvelles demandes d'accès à une ressource et résout du même coup les problèmes d'exclusion mutuelle. Toutefois, cette technique montre vite ses limites puisque l'utilisation d'une ressource peut être relativement longue et il n'est pas toujours souhaitable d'interdire la préemption (conséquence du masquage des interruptions) sur une telle durée. C'est pourquoi on lui préfère le plus souvent l'utilisation de sémaphores, qui permettent d'implémenter toutes sortes de politique d'accès à une ressource comme par exemple la politique FIFO (First In First Out) ou encore la politique de priorités fixes. Une fois l'exclusion assurée, il reste à la charge de l'exécutif temps réel de vérifier qu'il n'y a pas de phénomène d'interblocage
- **Gestion du temps** : Le temps est utilisé ici comme une horloge absolue pour cadencer le système. Nous utilisons traditionnellement une discrétisation du temps permettant au processeur d'effectuer une action atomique minimale au vu des instructions de l'application. La notion de temps intégrée dans un exécutif Temps Réel doit ainsi permettre de satisfaire plusieurs exigences [BUR90] :

– l'accessibilité du temps courant pour permettre la mesure du temps écoulé.

– la mise en attente d'une tâche pendant une durée finie.

– la définition d'une minuterie ou timer pour la détection par exemple de la non occurrence d'un évènement attendu.

- **Gestion des interruptions et de la mémoire** : La gestion des interruptions doit permettre de tenir compte toutes les sollicitations matérielles et logicielles. Nous utilisons un service de routines d'interruption (ISR) permettant d'associer un traitement à chaque exécution. La durée de chaque routine doit être la plus courte possible puisque les routines s'exécutent de manière atomique (les interruptions sont masquées durant leurs exécutions). La gestion de la mémoire peut être faite suivant deux modèles : soit l'exécutif et les tâches ont chacun une zone de mémoire réservée, soit chaque tâche ainsi que l'exécutif possèdent une zone mémoire séparée et protégée.

Toutes ces fonctions de l'exécutif Temps réel existent sous forme de primitives ou routines élémentaires dont la plupart possèdent des bribes atomiques, c'est à dire ne pouvant pas être interrompues par la gestion des interruptions matérielles. Ces portions ininterruptibles engendrent des retards dans la gestion des évènements qu'ils soient logiciels ou matériels. Pour assurer un service optimal aux traitements des tâches, il faut réduire ces portions au minimum. C'est justement l'un des critères d'évaluation des exécutifs Temps Réel du marché (ou RTOS), ce qui les différentient des systèmes d'exploitation classiques. Les RTOS assurent ainsi une borne temporelle pour chacune des primitives temps réel qu'elles proposent. Parmi ces RTOS [YVO05], nous pouvons citer par exemple Osek/VDX, Vxworks, RTEMS, Linux RT.

5. Quantification du temps

L'architecture logicielle des applications temps réel permet d'identifier le traitement d'un évènement à une tâche. Nous avons vu que ce traitement doit intervenir dans des délais appropriés. Il faut donc être à même de vérifier que le respect des contraintes temporelles est bien assurée. Pour cela, nous devons introduire des indications temporelles quantitatives permettant par exemple d'exprimer les délais à respecter. Ceci est mis en oeuvre par la modélisation temporelle des tâches. De plus, il est nécessaire de préciser la façon dont ces délais doivent être

pris en compte. Nous devons, en d'autres termes, préciser la qualité de service attendue pour l'évaluation de l'application temps réel.

5.1. Les Tâches en Temps Réel

Il existe trois types de tâches en Temps Réel qui diffèrent par leurs caractéristiques temporelles. Les tâches dites Périodiques sont la plupart du temps stimulées par l'Horloge Temps Réel (HTR) de l'exécutif temps réel de façon à assurer une activité régulière, par exemple lors de l'acquisition de données (comme dans le cas d'une lecture échantillonnée d'un signal continu) ou la génération périodique d'évènements. Les tâches apériodiques sont quant à elles activées de façon aléatoire en fonction par exemple d'évènement aléatoire.

Nous pouvons noter qu'il existe une sous famille de ce type de tâches qui est la famille des tâches sporadiques pour lesquelles une durée minimale sépare deux occurrences successives de l'évènement déclencheur. Enfin les tâches cycliques [HAN95] sont très proches des tâches périodiques à la différence prêt que leur activation n'est pas liée à l'Horloge Temps Réel, ce qui induit une périodicité approximative. La durée séparant deux activations successives d'une tâche périodique est constante alors qu'elle appartient à un intervalle [Pmin, Pmax] pour les tâches cycliques. Nous ne nous intéresserons par la suite qu'aux tâches périodiques et apériodiques.

Nous utiliserons le terme de tâche pour désigner le programme informatique compilé qui sera exécuté sur le processeur du système.

5.1.1. Tâche périodique

Le modèle de tâche périodique représente les tâches activées à intervalles réguliers (constants).

Soit une tâche périodique Ti alors Ti est modélisée par les quatre paramètres temporels :

- Ri la date à laquelle la première instance de Ti est activée
- Ci la pire durée d'exécution (ou charge maximale) de Ti
- Di le délai critique (ou échéance relative) associé à Ti
- Pi la période de la tâche Ti.

5.1.2. Tâche apériodique

Les tâches apériodiques ont pour origine des activations de deux types : elles peuvent provenir d'une intervention extérieure inattendue (comme une intervention humaine sur le procédé par exemple), ou provenir de l'application elle-même lorsque par exemple une tâche périodique chargée de faire de l'acquisition détecte une valeur inattendue nécessitant un traitement ponctuel spécifique. Leur importance dépend de la criticité de l'information qu'elles doivent traiter.

Les évènements déclencheurs étant en tout état de cause imprévisibles, les tâches apériodiques sont des tâches dont la fréquence d'activation est totalement aléatoire. Les paramètres du modèle précédent comme les dates de réveil et périodes n'ont par conséquent plus lieu d'être ici. Par contre, une tâche apériodique possède bien une durée d'exécution bornée par un WCET (Worst-Case Execution Time) Ci et éventuellement un délai critique Di pour s'assurer de son exécution dans un temps borné.

Les tâches sporadiques possèdent un paramètre supplémentaire permettant de définir un intervalle minimal entre deux activations successives Cet intervalle minimal est généralement assimilé à son délai critique. Il existe de nombreuses définitions et particularités sur ce type de tâches.

Soit une tâche Ti.

- Si Ti est une tâche apériodique, alors Ti est modélisée par un unique paramètre temporel : Ci sa pire durée d'exécution.
- Si Ti est une tâche sporadique, alors Ti est modélisée dans le cas général par :(Ci, Di, Tsi) où Tsi correspond à l'intervalle de temps minimum séparant deux activations successives et Di au délai critique.

5.2. Qualité de service

Dans le paragraphe précédent, nous avons défini les paramètres temporels accordés à une tâche. Nous indiquons maintenant la nature des contraintes qu'ils engendrent.

En effet, les systèmes temps Réel n'ont pas tous le même degré d'exigence vis à vis de ces critères. Si nous considérons un système critique embarqué dans un avion, il est vital que les tâches d'un tel système aient des temps de réponse rigoureusement contrôlés, inférieurs systématiquement à une borne fixée (exprimée par le délai critique des tâches). Au contraire, un

attardement de réaction (par rapport aux bornes fixées par les concepteurs qui correspondent à un fonctionnement optimal) lors de la compression vidéo n'entraîne aucune catastrophe, ni même de perturbation sensible si ce retard n'intervient pas trop souvent. Cette constatation permet de définir des classes de systèmes temps réel suivant le degré de criticité de leur qualité de service. On distingue ainsi 3 familles de systèmes temps réel suivant la rigidité des contraintes temporelles qui leurs sont imposées :

- Les Systèmes Temps Réel à Contraintes Strictes. Ce type de système impose que toutes les contraintes temporelles soient impérativement respectées
- Les Systèmes Temps Réel à Contraintes Souples. À l'opposé de la classe précédente, un non respect d'une échéance n'entraîne pas la défaillance du système. Ces dépassements sont donc tolérés mais entraîne des perturbations qu'il faudra alors minimiser.
- Les Systèmes Temps Réel à Contraintes Mixtes. Ces derniers sont soumis à la fois aux exigences des systèmes à contraintes strictes pour certaines tâches et à celles des systèmes à contraintes souples pour d'autres.

5.3. Ordonnancement et validation

L'exécutif temps réel est constitué d'une base communément appelée ordonnanceur, encapsulé par des agences qui offrent aux tâches les services requis pour leurs synchronisations, communications, temporisations…

Le problème de l'ordonnancement, sur lequel repose la validation de l'application, consiste à définir une politique d'attribution du processeur (et des ressources) qui assure qu'aucune faute temporelle ne sera commise, c'est-à-dire qu'aucune tâche ne terminera l'une quelconque de ses instances après l'échéance de celle-ci.

- L'ordonnancement en ligne, où un algorithme est implanté au niveau de l'ordonnanceur, les décisions d'ordonnancement étant prises au cours de l'exécution de l'application chaque fois qu'une nouvelle instance de tâches est activée ou qu'une instance termine.
- L'ordonnancement hors ligne , qui est calculé sur l'ensemble des tâches avant l'exécution effective de l'application, la séquence ainsi produite est chargée dans une table qui sera utilisée par le répartiteur. Notons que l'utilisation d'un ordonnancement hors ligne permet d'éviter la surcharge processeur liée à l'exécution de l'algorithme d'ordonnancement. En

contre partie, un ordonnancement en ligne est plus souple, en particulier en cas de reconfiguration de l'application, ou en cas de prise en compte de tâches sporadiques ou apériodiques.

6. Conclusion

L'objectif de ce chapitre était de présenter le cadre scientifique sur lequel nous nous sommes reposés pour mener notre travail. Nous avons vu la spécificité des systèmes temps réel vis à vis des systèmes informatiques classiques. Nous avons mis en évidence l'importance de la quantification du temps dans les applications temps réel et nous avons alors exhibé les différents modèles temporels de tâches qui les composent ainsi que les différentes mesures pouvant être associées aux tâches. Nous nous sommes ensuite intéressés aux problèmes de l'ordonnancement de tâches et avons énuméré les classes de problèmes d'ordonnancement les plus importants. Nous avons ainsi pu mettre en évidence les difficultés d'ordonnançabilité que peut engendrer un modèle. Enfin et face à la montée croissante des systèmes temps réel, notamment avec le développement des systèmes temps réel et des systèmes sur puces appelés SoC, de nouveaux besoins sont apparus : des exigences de sûreté de fonctionnement peuvent être exigées lors de l'exécution des applications, dans des domaines aussi divers que l'automobile, l'avionique, l'armement, et la transmission de flux multimédias. Pour répondre à ces besoins des approches formelles sont apparues pour spécifier ces systèmes. Nous présenterons en détail ces méthodes dans le chapitre suivant.

C H A P I T R E

2

Approches de modélisation des Systèmes temps réel

1. Introduction.

L'étude des approches de modélisation des STRE (Système temps réel embarqués ou RTES : Real Time Embeded System) est faite selon des niveaux d'abstraction plus au moins variés. Certains travaux montrent qu'on peut appliquer des techniques de génie logiciel que nous présentons au cours de ce chapitre. Nous nous intéressons plus particulièrement à la dimension temporelle de ces méthodes. Cette présentation est centrée d'avantage sur des aspects conceptuels plutôt que sur des aspects techniques de codage. Nous citons ROOM (Real-Time Object-Oriented Modeling), SDL (Specification and Description Language), ADL (Architecture Description Language) et l'approche orientée objet, et plus précisément la notion de profil UML [UML04] où l'accent a été mise.

La plupart des profils UML souffre de deux limitations majeures : ils ne prennent pas en charge intrinsèquement la modélisation de l'RTOS, et leurs performances peuvent s'en trouver détériorées en faveur de la modularité de leur conception. Ces limitations pourraient laisser penser qu'il est peu envisageable de construire des applications ayant des contraintes temporelles.

2. ROOM

Il s'agit d'un langage de modélisation visuel associé à une sémantique formelle [ROO96], Il a été développé par la société ObjecTime. Il est optimisé pour la spécification, la visualisation, la documentation, l'automatisation et la construction de systèmes temps réels potentiellement distribués, complexes et « orientés-événement ».

ROOM n'est pas aussi générique de façon qu'il soit parfois utilisable dans certaines problématiques tandis que pour d'autres, il est préférable d'utiliser d'autres méthodes.

ROOM travaille essentiellement avec des Interfaces. Celles-ci sont réifiées dans des classes Port et l'interaction complexe entre les objets est transposée dans des classes Protocol qui permettent d'arbitrer le comportement entre Ports. A l'inverse des interfaces UML, les classes

L'un des bénéfices les plus intéressants de ROOM est qu'il modélise les deux côtés d'une interface, tant le client que le serveur. Par ailleurs, l'utilisation de classes Port, Connector et Protocol permet une très bonne définition des interfaces complexes. Les classes Protocol sont quasi exclusivement modélisées par des statecharts, ce qui facilite l'utilisation des pré et post

conditions des services. De plus, le fait que l'on puisse abstraire les interfaces (UML) en Port et classes Protocol, permet de leur donner un état et des attributs, ce qui les rend donc plus puissants.

Toutefois, les associations entre les classes sémantiques (Capsules) passent par l'intermédiaire d'un ensemble d'autres classes. Une telle structure peut compliquer inutilement les relations entre classes ayant une sémantique simple. Aussi, l'usage de ROOM doit être circonstancié. Il peut être utilisé lorsque des classes ont une sémantique qui est relativement importante et complexe. ROOM est particulièrement approprié quand l'interaction de quelques objets importants est complexe et requiert des significations spéciales pour contrôler et arbitrer des choix.

3. SDL

Les systèmes temps réel s'appuient sur un système d'exploitation temps réel (RTOS) dont l'élément structurant de base est la tâche. Plusieurs tâches s'exécutent en parallèle pour réaliser les fonctions de base qui sont regroupées ensuite pour réaliser les fonctions les plus complexes et ainsi de suite jusqu'à couvrir toute l'application. Il est rapidement apparu que le langage SDL [SDL04], qui permet d'organiser son application en regroupant les tâches en blocs fonctionnels qui, eux-mêmes, peuvent être regroupés en blocs de plus haut niveau, il est considéré comme un bon moyen de représenter graphiquement l'architecture de n'importe quel système temps réel.

D'un point de vue statique, une interface se définit par un format d'échanges basé sur des données structurées tant dis que d'un point de vue dynamique et en particulier par l'écriture d'un scénario qui décrit le séquencement des échanges. Pour la représentation de ces interfaces, le SDL est relativement bien adapté. En effet, les signaux SDL accompagnés de paramètres typés basés sur les données du langage (appelées Types de données abstraits ou ADT, Abstract data types) permettent une description complète d'une interface statique.

D'un autre point important: les systèmes temps réel sont basés sur l'exécution en parallèle de tâches indépendantes. Il est donc important dans ce contexte de ne pas gaspiller du temps CPU (Central Processing Unit) lorsqu'une tâche n'a rien à faire. Ceci a conduit la plupart des applications temps réel à se baser sur des machines à états finis, dans lesquelles le principe de base est de se mettre en attente sur un objet de l'RTOS, comme une file d'attente de messages, dès que la tâche a terminé son action. Ici, les machines à états finis du langage SDL sont parfaites

pour représenter graphiquement ce type de comportement. Enfin, depuis sa version 92, le SDL est orienté objet à tous les niveaux de la représentation graphique, ce qui permet de construire des librairies de composants logiciels spécialisés, adaptés au marché du temps réel. Le SDL a dû s'adapter aux contraintes du temps réel. Sur le papier, le SDL apparaît donc comme un langage parfait pour la spécification et la conception des systèmes temps réel. Alors que la réalité technique est tout autre : Comme premier problème, les ADT ne sont pas adaptés aux impératifs de la conception d'un système temps réel. En effet, leur syntaxe de manipulation a été définie pour signaler des protocoles, non pour les concevoir. Les développeurs sont alors frustrés de ne pas avoir la précision qu'ils avaient avec des langages de programmation classique comme le C. De plus, l'intégration de code existant est difficile car il faut réaliser une passerelle entre les types de données SDL et les types de données C ou C++. Autre difficulté, il n'existe pas de compilateur SDL natif ou croisé sur le marché. Par conséquent, une phase intermédiaire de génération de code en C serait nécessaire pour implémenter le système SDL sur cible.

4. Réseaux de Pétri

Les réseaux de Petri permettent d'étudier des systèmes dynamiques complexes quoiqu'ils restent autonomes de l'architecture du système. Ils ont été proposés en 1962 par Carl Adam PETRI [PET62]. Ils sont maintenant utilisés pour spécifier, modéliser et comprendre les systèmes (au sens informatique) dans lesquels plusieurs processus sont interdépendants. Ils constituent un outil graphique et mathématique de modélisation. Dans le cas des STRE, ils restent restreints pour la modélisation de l'aspect concurrence et ordonnancement des tâches.

[DEL03] présente l'approche UML/PNO (Unified Modelling Language with Petri Net Objects) pour la spécification de systèmes temps réel. La méthode propose d'enrichir la description semi-formelle UML du système par une modélisation formelle basée sur les réseaux de Petri. Les concepts UML de sous-systèmes et d'interfaces ont été étendus afin d'améliorer la description de la vue système en termes de structuration, gestion de projets et organisation de la modélisation. L'objectif est également d'adapter la méthode de modélisation système à une approche « orientée composant » pour le temps réel. Le concept « d'objet composé » permet d'intégrer des spécificités temps réel au sein du composant (protocole de communication, contrainte temporelle, effet observable).

Le comportement des objets est spécifié à l'aide des réseaux de Petri à places et transitions stochastiques temporelles afin de permettre la validation et la vérification du système en cours de spécification. L'approche de validation propose des traductions semi-automatiques des diagrammes UML en réseaux de Petri. Les techniques classiques de simulation et d'évaluation de performances peuvent alors être appliquées. Ces traductions dévoilent l'avantage de rassembler, sur un même modèle à réseau de Petri, tous les aspects dynamiques du composant apparaissant dans différents diagrammes UML et d'examiner de ce fait la cohérence de son comportement et de son utilisation. La vérification utilise les techniques d'analyse formelle, basées sur l'utilisation conjointe du graphe de classes et de la logique linéaire.

Pour [PAI06], il considère le problème de l'ordonnancement hors ligne d'applications Temps Réel multitâches dans le contexte où les tâches peuvent comporter des instructions conditionnelles. Il redéfinit le modèle temporel de tâches pour prendre en compte explicitement les instructions conditionnelles. Il reformule le problème de l'ordonnançabilité pour des tâches indépendantes et met en évidence l'ordonnançabilité globale et locale. Il étudie l'impact de la présence d'instructions conditionnelles sur les durées nécessaires de simulation. Il propose une méthode d'analyse d'ordonnançabilité fondée sur une modélisation par réseaux de Petri. L'ajout de tâches conditionnelles dans cette modélisation permet d'intégrer explicitement les différents comportements d'exécution des tâches et de prendre en charge l'activation des tâches sporadiques.

5. ADL

Le domaine des systèmes temps réel affiche des besoins qui justifient actuellement une réelle réflexion sur l'approche de conception architecturale : organisation complexe (présence de fonctionnalités multiples interdépendantes), architectures matérielles réparties, présence de contraintes non-fonctionnelles qui lient intimement éléments logiciels et matériels, utilisation optimisée des ressources, reconfigurabilité dynamique, prédictibilité et donc nécessité de vérification a priori et au plus vite dans le processus de développement, etc. De plus, la généralisation de l'utilisation des systèmes temps réel embarqués dans des domaines comme l'automobile ou l'avionique (où les produits sont déclinés en gamme et sont construits par assemblage de sous-systèmes fournis par différents équipementiers) fait émerger des exigences nouvelles de flexibilité, réutilisabilité, portabilité, interopérabilité, etc. Les ADLs [ANN05]

constituent une classe de langages offrant des abstractions pour la description « gros grain » des systèmes logiciels. De multiples langages appartiennent à cette catégorie, langages qui, pour certains, diffèrent de manière majeure ne serait-ce que par leur syntaxe, leur sémantique, leur expressivité et les buts qu'ils visent. En proposant une définition unique, précise et consensuelle est alors un problème délicat.

L'objectif d'un langage de description d'architecture est avant tout d'exprimer les relations entre les composants de l'application. Ces interactions deviennent rapidement un point dur du développement dans le cadre d'applications réparties ou modulaires car elles sont construites par assemblage de composants dont les interfaces doivent être compatibles. C'est pour cette raison que le développement de l'avionique modulaire a conduit à la définition d'un langage de description d'architecture AADL appelé initialement (Avionics Architecture Description Language) et ensuite renommé Architecture Analysis & Design Language, il est particulièrement précis sur la définition de l'interface entre les composants de l'application, mais également sur la description de l'environnement cible et de la manière dont les composants sont déployés. Cet AADL émergeant est un ADL développé pour répondre aux besoins spéciaux des systèmes embarqués temps-réel, il évoque explicitement des points méthodologiques comme la génération automatique de code. L'objectif du code produit est double : relier entre eux les composants de la spécification en fonction de leur interface et déployer ces composants sur un environnement d'exécution cible. AADL constitue une approche orientée avant tout sur le déploiement et la spécification de l'architecture de l'application afin de permettre une expression claire des liens de communications entre ses composants. Seules les interfaces des différents modules sont spécifiées. Les aspects comportementaux ou liés au contenu des données échangés sont décorélés de la spécification AADL et traités dans un autre formalisme. Cette ouverture vers d'autres formalismes de spécification est une fonctionnalité très intéressante. Ainsi AADL propose un mécanisme d'extension du langage par un système d'annexes, et la possibilité de relier entre eux des composants écrits dans d'autres langages. Ainsi chaque composant peut être développé dans le formalisme le plus adapté, et intégré à l'application en décrivant son interface en AADL.

Dans les ADL les interfaces des composants sont décrites de manière syntaxique, avec très peu de sémantique associée. On reproche alors aux ADL le manque de sémantique commune des modèles et leurs objectifs de conception différents qui limitent les capacités d'interaction des langages et de leurs outils.

6. UML et le temps réel

UML fournit les fondements pour spécifier, construire, visualiser et décrire les artefacts d'un système logiciel. Pour cela, UML se base sur une sémantique précise et sur une notation graphique expressive. Il définit des concepts de base et offre également des mécanismes d'extension de ces concepts.

6.1. Modélisation comportementale avec UML

Du point de vue comportemental, UML propose principalement trois constructions : les machines d'états, les diagrammes d'activité et un langage d'action (Action).

6.1.1. Les diagrammes d'activité

Les diagrammes d'activité servent à faciliter la modélisation de traitements complexes en termes de flots de contrôle et de flots d'objets entre les différents constituants de l'activité.

La sémantique associée aux diagrammes d'activité repose sur une circulation de jetons, proche de celle rencontré dans les réseaux de Pétri. Un jeton modélise une donnée ou un objet. Sa circulation dans le réseau est conditionnée par les éléments de contrôle (arcs ou noeuds). Ces éléments permettent d'exprimer des notions de parallélisme et de synchronisation.

On distingue trois types de noeuds. Les noeuds d'action transforment les flux de données/contrôle d'entrée en flux de données/contrôle de sortie. Ces derniers sont alors les entrées d'autres actions. Les noeuds de contrôle définissent les règles de circulation des jetons à travers le graphe. Les noeuds objets servent à stocker temporairement des données ou des objets. Pour connecter ces noeuds, il existe deux types d'arcs : les arcs de flux de contrôle et les arcs de flux d'objet. Les premiers synchronisent le début d'une action (destination de l'arc) avec la fin d'une autre action (origine de l'arc). Les arcs de flux d'objet permettent de faire passer des valeurs entre deux noeuds.

6.1.2. Les machines d'états

Attaché à une classe ou à un cas d'utilisation, le diagramme d'états transitions présente une classe par rapport à ses états possibles et aux transitions qui le font évoluer. Il permet de spécifier ce que doit faire l'objet en réponse aux événements (ou traitements) qui lui sont appliqués.

Les machines d'états offrent de nombreux concepts, tels que la notion d'état hiérarchique, composite, historique et organisé par noeuds de branches qui, combinés, couvrent la plupart des formalismes sur la notion d'état. La description de la sémantique des machines d'états [NIZ06] est de type opérationnel et repose sur une machine d'exécution hypothétique qui présente les trois caractéristiques suivantes :

- une file d'attente d'événements qui sert à stocker les instances d'événements entrantes en attendant de les consommer.
- une politique de choix des événements qui détermine l'ordre d'extraction des occurrences d'événement contenues dans la file d'attente : Lors d'une exécution, une machine à états accède à un pool d'évènements géré par l'objet contexte de la machine. En fonction de l'état courant de la machine et de l'ensemble d'évènements pertinents contenus dans le pool (c'est-à-dire ceux pouvant déclencher une transition à partir de l'état courant), la politique de sélection des évènements détermine un ordre pour l'extraction des évènements du pool, et offre la possibilité de mettre en œuvre différentes politiques de gestion des priorités
- un processeur à événement qui exécute les traitements associés aux événements en respectant la sémantique des machines d'états transitions de UML et, en particulier, l'hypothèse d'exécution (***Run-to-Completion***). Les événements sont dépilés un par un et consommés par une machine d'états transitions. L'ordre dans lequel ils sont dépilés n'est pas précisé dans UML, cela constitue un point de variation sémantique. La sémantique d'exécution des événements est basée sur l'hypothèse dite de (Run-to-Completion). Cela signifie qu'un événement ne peut être dépilé puis consommé que lorsque le traitement de l'événement précédent est achevé.

Une machine à états possède une ou plusieurs régions, elles-mêmes composées de sommets (états ou pseudo-états) et de transitions reliant ces sommets. Les transitions sont gardées par une contrainte, et déclenchées par un trigger référençant un évènement déclencheur. Tirer une transition provoque l'exécution du comportement (potentiellement) associé, et la modification de l'état courant de la région, de l'état source à l'état cible de la transition.

Pour la modélisation du temps au niveau du diagramme d'états transitions, UML définit un événement spécifique appelé ***TimeEvent***. Il sert à modéliser l'expiration d'une échéance qui peut être relative ou absolue :

- un événement dénotant le passage d'une quantité de temps suite à l'entrée dans l'état contenant la transition est noté avec le mot-clé ***after*** suivi d'une expression de type ***TimeExpression*** qui donne la valeur temporelle de l'événement.
- un événement dénotant l'occurrence d'une date absolue est noté via le mot-clé ***when*** suivi d'une date absolue de type ***Time***.

Le tableau 1 décrit trois extraits de machine à états transitions illustrant l'utilisation possible des événements temporels de UML. Dans les trois cas, lorsque le temporisateur armé arrive à échéance, il génère un événement qui est stocké comme tout autre événement dans la file d'attente associée à la machine d'états. Si celle-ci est dans l'état **S**

Modélisation des évènements temporels	**Description**
S After (10ms) /'procedure' …..	L'évènement est généré 10 ms après la date d'entrée dans l'état S.
S after(10 ms after the exit of S2) *'Procédure'* …..	L'évènement est généré 10 ms après la date de sortie de l'état S.
S When (1er Janvier 2000, 0h00) *'Procédure'* …..	L'évènement est généré le 1er Janvier 2000 à 0h00.

Tableau 1*: *Exemple de spécification d'événements temporels.

6.1.3. Le langage d'action

UML définit le concept d'*Action* comme étant l'unité fondamentale de spécification comportementale permettant à des modèles UML d'être complètement exécutable. Le principe repose sur le fait que les actions échangent des flux de contrôle et de données via des fiches d'entrée et de sortie.

En effet, UML2.0 définit un package particulier appelé *Actions* [DUB05] qui définit en détails comment modéliser toutes les actions d'une application afin d'obtenir un modèle exécutable. Les actions sont les entités comportementales de base permettant la spécification de modèles UML

exécutables. Les actions échangent des flots de contrôle et des flots de données à travers des *InputPins* et des *OutputPins*. Le standard UML s'attache à définir la sémantique des actions en les regroupant en quatre paquetages

- Le paquetage BasicActions définit les actions d'appel d'opérations, d'envois de signaux et d'invocation de comportements.
- Le paquetage IntermediateActions définit des actions d'invocations (diffusion de signaux et envois d'objets qui ne sont pas des signaux), des actions de lecture et d'écriture d'objets, de caractéristiques structurelles et de liens.
- Le paquetage CompleteActions définit des actions traitant de la relation entre les objets, les classes et les liens d'objets ainsi que des actions de gestion des événements tels que les acceptations d'appels d'opération.
- Le paquetage StructuredActions définit les actions opérant dans le cadre d'activités. Ces actions concernent la manipulation de variables et la gestion des exceptions.

Au travers des précédents sous paquetages, le standard UML2.0 s'est uniquement attaché à définir une syntaxe abstraite et une sémantique pour le Langage d'Action UML. Mais, cette norme est inutilisable telle quelle est, car aucun langage de surface (ou syntaxe concrète) n'est proposée qui satisfasse la sémantique.

Le modèle d'activités de UML 2.0 est organisé en différents paquetages reflétant les différents niveaux de sémantique offerts :

- Le paquetage *FundamentalActivities* définit les activités comme noeuds contenant des actions. Ce niveau sémantique est partagé par les activités basiques (*basic activities*) et les activités structurées (*structured activities*).
- Le paquetage *BasicActivities* offre la spécification des activités avec des flots de contrôle et de données entre les actions.
- Le paquetage *IntermediateActivities* définit les activités offrants les flots de contrôle et de données concurrents. Ce niveau de sémantique permet la modélisation de réseaux de Pétri classiques.

- Le paquetage *CompleteActivities* permet la définition d'activités avec des constructions évoluées telles que les transitions pondérées ou encore le « streaming » de données.
- Le paquetage *StructuredActivities* permet la définition d'activités constituées de constructions classiques de programmation structurée comme les boucles ou les branchements conditionnels.
- Le paquetage *CompleteStructuredActivities* permet la définition d'activités avec des flots de données en sortie des boucles ou des branchements conditionnels.
- Le paquetage *ExtraStructuredActivities* permet la définition d'activités contenant des exceptions ou des invocations de comportement sur des ensembles de valeurs.

6.1.4. Diagramme de séquence

Le diagramme de séquence permet de décrire une interaction qui est elle-même un ensemble de messages entre des instances en vue de réaliser l'opération ou le résultat désiré.

Le diagramme de séquence associe à chacun des objets impliqués dans une interaction une ligne de vie verticale représentant le temps (le temps s'écoule de haut en bas) et permettant d'identifier explicitement la séquence et l'ordre des messages émis et reçus par les objets. L'ordre est partiel par rapport à tout le système. Lorsqu'on désire indiquer qu'un message déclenche un traitement particulier dans l'objet, on représente ce traitement avec un petit rectangle vertical le long de sa ligne de vie.

6.2. Profil UML

A sa création, UML avait pour ambition de se positionner comme langage de modélisation couvrant l'ensemble des domaines du logiciel, comme les bases de données, les systèmes embarqués ou les systèmes de gestion. UML s'est ainsi imposé sur certains de ces domaines et continue sa pénétration dans les autres, même si parfois des difficultés sont rencontrées. De plus, au fur et à mesure de l'adoption de UML, son champ d'investigation s'est ouvert à d'autres domaines tels que l'électronique et, plus généralement, à l'ingénierie système.

Parce qu'un même et unique langage ne pourrait pas répondre à toutes les spécificités de chaque domaine, UML propose des possibilités de spécialisation permettant d'adapter le langage à des besoins particuliers. Cet objectif peut être atteint principalement par deux moyens : les points

ouverts de variation sémantique et les mécanismes d'extension. Les mécanismes d'extension visent à adapter UML à des besoins spécifiques. Cela peut se faire via trois concepts particuliers de UML : les stéréotypes, les valeurs étiquetées «Tagged values» et les contraintes. L'adjonction d'un stéréotype peut être vue comme l'ajout d'une nouvelle méta classe au méta-modèle de UML, c'est-à-dire comme l'ajout d'un nouveau mot au vocabulaire de base proposé par UML. Un stéréotype est défini dans le contexte d'un élément existant du méta-modèle de UML et « hérite » ainsi de ses caractéristiques (attributs, opérations, relations, …). Lors de la définition d'un stéréotype, une nouvelle représentation graphique peut lui être attachée si besoin est. Associées à un stéréotype, il est également possible de définir des valeurs étiquetées. Celles-ci peuvent être vues comme étant de nouveaux méta attributs liées à un stéréotype ; elles permettent ainsi d'en définir des nouvelles caractéristiques. De même, il peut être utile de définir un ensemble de contraintes qui, associées à un stéréotype, clarifieront la sémantique de façon formelle, en utilisant par exemple le langage Object Constraint Language(OCL). En vue d'organiser l'éventuelle prolifération de stéréotypes, valeurs étiquetées et autres contraintes, le concept de profil [UML04] a été introduit afin de regrouper sous une même bannière un ensemble d'extensions de UML défini de façon à faciliter la modélisation d'un problème particulier.

Face à des contraintes de productivité et de concurrence de plus en plus forte, les industriels du secteur des systèmes embarqués temps réel se sont tournés vers les technologies orientées objets et/ou composants. Ils sont ainsi naturellement arrivés à adopter UML et l'ingénierie dirigée par les modèles (IDM) parce que :

- La spécification des systèmes temps réel embarqués implique différents points de vue (ex. fonctionnel, temps réel, tolérance aux fautes…) et il est difficile de distinguer les aspects génériques des produits des aspects particuliers de leur implantation. En effet, les contraintes de réalisation de telles applications, souvent dictées par les limites des ressources disponibles, sont telles qu'il est difficile d'en faire abstraction lors de la spécification d'une application.
- Les options d'implantation visées peuvent varier considérablement ; différents modèles d'exécution peuvent être envisagés pour un même modèle en fonction des contraintes de réalisation particulières (ex. modèle multi-tâches avec Real Time Operating System,

modèles d'exécution synchrone…). De plus, les options d'implantation peuvent également reposer sur des solutions propriétaires compliquant les éventuels portages vers d'autres cibles matérielles.

- La performance de tels systèmes est un problème critique qui va souvent à l'encontre des techniques standards d'encapsulation logicielle. Les optimisations peuvent alors mener à des entrelacements du code fonctionnel et temps réel qui devient alors un facteur pénalisant en terme de maintenabilité des applications.
- Le test et la validation des applications temps réel embarquées sont des activités critiques qui nécessitent la mise en place de modèles et d'outils d'analyse sophistiqués spécifiques.
- Enfin le domaine du développement temps réel embarqué nécessite des développeurs à haut niveau d'expertise en conception, implantation et validation.

Pour répondre aux problèmes très spécifiques du domaine temps réel, nous présente les profils UML suivants : le profil « Rapid Object-Oriented Process for Embedded Systems » (ROPES), le profil OMEGA, le profil « Scheduling, Performance and Timing » (SPT) et le profil «Quality of Services & Faults Tholérance» (QoS&FT), le profil ACCORD /UML, le profil GASPARD (Graphical Array Specification for Parallel and Distributed Computing) et le profil Modeling and Analysis of Real-Time and Embedded systems (MARTES).

6.2.1. ROPES

Le processus ROPES [ROP04] est basé sur un cycle de vie du développement itératif qui utilise le standard UML et qui encourage la génération automatique de code et ce afin de parvenir rapidement à des prototypes à valider.

Ce processus de développement est divisé en quatre grandes phases :

- Analyse : Cette phase permet d'identifier toutes les caractéristiques du système à développer. Celle-ci peut être découpée en trois sous-phases qui ont chacune leurs spécificités. Les diagrammes utilisés pour cette phase peuvent donc être choisis parmi les propositions suivantes : texte en langue naturelle, diagrammes de Use Case, Diagrammes de séquences, Diagrammes d'états

- Design : Tandis que la phase d'analyse identifie la logique d'un système, le design propose une solution particulière unique optimale. Trois sous-phases composent cette étape :
  - Le design architectural définit les décisions stratégiques du design qui affecteront les composants software tel que le modèle de concurrence, la distribution des composants.
  - Le design mécaniste ajoute la liaison entre les composants.
  - Le design détaillé définit la structure interne et le comportement de chaque classe, composant, ...

  La phase de design consiste majoritairement en l'application de « design patterns » au modèle logique établi à la fin de l'étape d'analyse
- Implémentation : Le passage de modèles conceptuels successivement affinés à du code concret et efficient caractérise cette phase. ROPES impose aussi qu'à ce stade des unités de test soient développées de manière à pouvoir valider chaque ensemble cohérent de code exécutable
- Test : Cette phase intègre tant les tests d'intégration que ceux de validation. Il est important de concevoir ces tests pour obtenir des résultats observables

6.2.2. OMEGA

Le profil OMEGA [OME05] spécialise une partie du profil dans le but de raffiner la description de contraintes temporelles sur le comportement du système et de formaliser la relation entre ces contraintes (annotations) et la sémantique du modèle fonctionnel. Il offre des moyens et une syntaxe concrète pour exprimer les contraintes temporelles à la fois de manière opérationnelle et de manière déclarative.

Les Concepts temporels opérationnels définis par le profil sont :

- Le temps courant a la même valeur partout dans le système et que cette valeur peut être accédée explicitement dans le langage d'actions, à travers l'expression *now*.
- Les temporisations sont des instances (d'une classe prédéfinie « Timer ») qui permettent de compter la progression du temps. Une temporisation peut être armée pour compter le

temps correspondant à une durée et elle génère un événement « TimeOut » après l'écoulement de la durée spécifiée, qui peut être utilisé pour déclencher une transition dans l'objet qui le détient.

- Les horloges (instances d'une classe prédéfinie « Clock ») sont des concepts empruntés aux automates temporisés. Ils permettent de mesurer et de consulter le temps écoulé depuis leur dernier démarrage.

Les concepts couverts par le profil OMEGA sont :

- La description de la structure, faite à travers des diagrammes de classes, incluant des relations d'héritage et d'associations.
- Un modèle particulier de concurrence, défini à partir des notions de classes actives et passives. Les primitives de communication entre objets sont les signaux asynchrones et les appels d'opérations.
- La description du comportement, faite à l'aide de machines à états (associées aux classes) et d'opérations. On distingue deux types d'opérations, les unes (*primitives*) décrites par une méthode, les autres « *triggered* » décrites par la machine à états de la classe.
- Les actions des transitions et des méthodes sont décrites dans un langage d'actions impératif conforme à la sémantique d'actions d'UML

Certes le profil OMEGA est très riche en termes de définition de propriétés non fonctionnelles du système mais présente un manque au niveau de définition d'un modèle d'ordonnancement et de vérification et de validation d'ordonnançabilité.

6.2.3. SPT

Il s'agit d'un profil UML concernant le temps, l'ordonnançabilité, et le temps. Ce profil [SPT02] a pour objectif d'augmenter la capacité de modélisation de la qualité de services liée au temps (échéance et durée). De plus, ce profil est capable de définir un système temps réel sous formes de ressources, de définir leurs propriétés temporelles et de modéliser leur déploiement sur une architecture cible. Pour la modélisation de ses contraintes, des stéréotypes sont mis en place comme par exemple deadline pour représenter une échéance simple. Ce profil est utilisé par Rashbody et son couplage à l'outil de validation RapidRMA qui permet l'analyse de

l'ordonnonçabilité du système. L'avantage du SPT est d'avoir été adopté par l'OMG et d'être capable de spécifier directement en UML des informations quantitatives.

Ce profil vise alors à définir un ensemble minimal de concepts nécessaires à la modélisation des aspects temps réel d'un système. Les concepts doivent aboutir à la description de modèles à partir desquels on doit être capable soit de produire une implantation ou des supports d'implantation, soit d'analyser le comportement temps réel d'une application. Pour ce faire, le profil SPT est constitué de deux sous-ensembles (paquetages) principaux : le paquetage des ressources générales et le paquetage d'analyse comportementale temps réel.

Au coeur du premier paquetage, on trouve les concepts de ressource et de qualité de service. Ce concept de qualité de service fournit une base pour la définition de contraintes temps réel qualitatives tels que : échéance, débit, temps d'exécution maximale, etc. Le paquetage d'analyse du profil SPT propose trois sous profils dédiés respectivement à l'ordonnançabilité classique, à l'étude de performance et à l'analyse d'ordonnançabilité dans le contexte de RT-CORBA mais non pas dans le contexte d'un RTOS.

6.2.4. GASPARD

Le profil GASPARD (Graphical Array Specification for Parallel and Distributed Computing) [GAS06] est un environnement intégré de développement pour la conception des SoC, et comme son nom l'indique, il est principalement destiné à la spécification des applications de traitement des signaux intensifs. Gaspard permet une modélisation de haut niveau pour la conception du logiciel et du matériel. Ce niveau d'abstraction permet l'utilisation des techniques de vérification avant tout prototypage. Il permet aussi de produire automatiquement une distribution et un ordonnancement de l'application sur l'architecture avec génération du code. Cet environnement assure donc la modélisation, la simulation, le test ainsi que la génération du code pour les SoC.

Gaspard est basé sur plusieurs niveaux d'abstractions. Dans le niveau le plus haut, le designer décrit son application logicielle et l'architecture matérielle sur laquelle va tourner l'application appropriée. IL s'étale sur six packages :

- package « component »
- package « factorization »
- package « application »

- package « hardwareArchitecture »
- package « association »
- package « control »

6.2.5. Profil QoS & FT

Le profil UML QoS & FT [QOS01] a une portée plus large que le profil SPT. Il permet à l'utilisateur de définir une variété plus large d'exigences de QoS et des propriétés (performance et tolérance aux pannes). La framework de ce profil supporte une catégorisation générale des différentes sortes de QoS. L'inclusion de la QoS est fixée lors de la conception et sera gérée dynamiquement. En outre, ce profil supporte l'intégration des différentes catégories de QoS pour modéliser la QoS des aspects du système.

Le framework de ce QoS fournit un méta-modèle pour définir les concepts de domaine et pour construire le profil QoS et un dépôt de spécifications QoS (nommé le Catalogue QoS).

Le tableau ci dessous présente une comparaison entre les deux profiles QoS&FT et SPT

Exigence	**Profil SPT**	**Profil QoS**
Annotation des processus	Faible	Bien défini
Permission à l'utilisateur d'ajouter des mesures	Non (mesures prédéfinies)	oui
Type pour des valeurs concernant le temps	Valeurs temps réel	aucun
Expressions pour définir des propriétés non fonctionnelles (NFPs) quantitatives	Partie du langage TVL	non
Expressions pour définir des contraintes	Limité	Riche avec OCL

Tableau 2*: *Comparaison entre les profils SPT et QoS

6.2.6. Profil MARTES

MARTE (Modeling and Analysis of Real-Time and Embedded Systems) [MAR06] vient des deux profils existants: SPT et QoS&FT. Ce profil permet la modélisation de systèmes embarqués réactifs, de contrôle/commande et leurs flux de données intensifs ainsi que leurs aspects logiciels et matériels. Ce profil assure la conformité avec le Profil QoS &FT pour modéliser la Qualité de Service et la tolérance aux pannes, il permet ainsi la spécification de contraintes non seulement en temps réel mais aussi d'autres caractéristiques QoS incorporées comme la consommation électrique et la taille de mémoire. MARTES offre aussi la possibilité de modélisation et d'analyse d'architectures à base de composants. Il possède la capacité de modéliser le temps asynchrone/causal, synchrone/horloge et réelle/Continue.

6.2.7. ACCORD/UML

Accord|UML est à la fois un cadre conceptuel et une méthodologie qui a pour but d'assister le développement d'applications temps réel en masquant le plus possible les aspects d'implantation pour permettre au développeur de se concentrer sur les aspects métiers du système (fonctionnalités, contraintes de performance…). Pour atteindre ce but, Accord|UML se place dans un contexte d'ingénierie des modèles, et s'appuie très largement sur des modèles

UML et des transformations automatiques pour le raffinement et l'enrichissement de ces modèles, ainsi que pour les aspects méthodologiques.

ACCORD/UML propose essentiellement deux concepts dédiés:

- Le stéréotype « RealTimeObject » (ou « RTO ») supportant la modélisation d'entités concurrentes et encapsulant les contrôles de concurrence et d'états. Il s'agit d'une extension de celui d'objet actif où il est possible d'attacher des contraintes temps réel aux traitements des messages effectués par les objets actifs de l'application. L'objet temps réel peut ainsi être vu comme un serveur de tâches. Chaque demande d'exécution d'une de ses opérations (un service) correspond alors à l'activation d'une tâche temps réel dans l'application. Un objet temps réel est capable de recevoir et d'émettre des messages qui peuvent être de deux natures différentes : un signal ou un appel d'opération.
- Un stéréotype « RealTimeFeature » (ou « RTF ») réifiant le concept de qualité de service du SPT pour le besoin de modélisation de contrainte temps-réel. Les différentes valeurs

étiquetées associées à ce concept permettant ainsi de modéliser des caractéristiques qualitatives temps réel telles que échéances, périodes, temps de début…

6.3. OCL et le temps réel

Le langage de contrainte d'objets [OCL04] a été présenté pour supporter la spécification de contraintes pour des diagrammes UML et est principalement employé pour formuler des invariants et des opérations pré et des post conditions. Bien que OCL soit également appliqué dans les diagrammes comportementaux, par exemple, le diagramme d'états transitions, ce n'est pas actuellement possible d'indiquer des contraintes au sujet des propriétés dynamiques de comportement et de synchronisation de tels diagrammes.

Le travail issu de [STE04] présente une extension de OCL avec des spécifications des contraintes états orientés temps délimités, de façon que OCL supporte non seulement des propriétés fonctionnelles du système mais aussi d'autres non fonctionnelles

7. Discussion

Il est clair d'après la description de la spécificité de chaque profil présenté précédemment, que l'accent est mis sur la description de l'architecture matérielle et de l'application. Ces profils sont fondés sur un niveau d'abstraction plus élevé que d'autres approches, ils visent également les applications à dominance flot de données et non pas celui de contrôle. Même dans le cas où ces travaux touchent l'aspect temporel, ils ne peuvent pas couvrir la modélisation de l'RTOS. On leur reproche ainsi le manque de sémantique temporelle et transitionnelle commune des modèles ainsi que la non disponibilité des outils qui les supportent.

En effet, ces travaux ne nous permettent pas encore de garantir la sûreté de fonctionnement du système, en d'autre terme son aspect déterminisme. Le modèle d'application n'est pas associé avec le modèle d'analyse et la liaison entre deux modèles doit être encore effectuée manuellement avec de nombreuses itérations. Ces modèles ne supportent pas suffisamment l'intégration des caractérisations temps réelles comme le temps d'exécution et les contraintes temps réelles et par conséquent ils ne tiennent pas compte de l'RTOS relatif à l'architecture et l'application considérées. Malgré cette absence d'intégration de l'RTOS dans les modèles, des travaux tels que [IME05] et [SAM06] ont démarré pour toucher cet aspect quoiqu'ils restent superficiels.

Les approches liées à la simulation ont besoin d'un temps de simulation assez long pour donner une vue de fonctionnement relativement fiable. Par ailleurs, les travaux basés sur l'exécution numérique impliquent toujours l'explosion combinatoire des états du système.

On a donc besoin dans un premier temps d'une méthodologie servant à spécifier le système et essentiellement ses propriétés non fonctionnelles et également à valider l'ordonnançabilité de ces systèmes temps réels.

Ensuite, pour amortir cette difficulté de conception, le recours à des CAD (Computer Aiding Design) TOOLS est une solution plus au moins efficace. Mais, abstraire le plus possible se voit une solution plus évidente. Ce ci pourrait être réalisé à travers le recours à une approche de méta-modélisation dans le but de modéliser les trois composantes principales du système : l'application, le modèle d'architecture et le système d'exploitation temps réel (RTOS). A partir du méta-modèle composé, on pourra vérifier formellement le comportement temporel du système.

L'approche MDA œuvre dans ce sens. En effet, l'approche MDA amène à un niveau d'abstraction encore plus haut à travers le principe de développement à base de modèles visant à promouvoir la conception de système indépendamment de toute plate-forme technologique logicielle. De plus, elle permet aussi la division orthogonale du système en plusieurs modèles de domaines, augmentant ainsi encore plus le niveau de réutilisation. Le « Plateforme Independant Model » (PIM) est le point de départ du processus MDA. Combiner une version spécifique de ciblage du cadre en temps réel avec une technologie de génération de code puissante et personnalisable est une façon très efficace et souple de traduire un PIM en PSM (Plateforme Specific Model). Cette façon permet également de recibler facilement le PIM vers un PSM différent. Cette méthode fournit un avantage en permettant de tester et de déboguer l'application au niveau de la conception ou du PIM sur l'hôte tout en prenant en compte les concepts et ressources de planification utilisés dans les environnements embarqués typiques ciblant soit un RTOS commercial soit un environnement de planification exclusif.

Pour résoudre tous ces problèmes, dans le travail du mastère, nous proposons une démarche offrant la possibilité de modéliser et de valider le système MDA sur les RTOS.

Les propriétés temps réels qui doivent être vérifiées lors de la modélisation sont :

- Le comportement dépendant du temps

- La prétention relative au temps sur l'environnement externe système comme le temps de réponse et le temps d'exécution d'une action
- Condition relative au temps tel que la date limite d'une action et la durée entre deux évènements.

8. Conclusion

Nous avons présenté dans ce chapitre une synthèse des approches de conception des systèmes temps réel. Nous avons notamment choisi d'étudier en détail des approches issues des ateliers de génie logiciel essentiellement les profils UML dédiés à la modélisation des STRE. Quelque soit le profil, nous avons pu identifier des caractéristiques communes : Manque de garantie de sûreté de fonctionnement et d'intégration de l'RTOS.

Toutefois, notre but n'est pas de recréer une nouvelle manière de modéliser des STRE, ni de proposer de solutions techniques pour les implémenter. Notre objectif est d'avantage l'intégration de la modélisation d'un RTOS avec l'application et l'architecture, notamment en se référant à l'ingénierie dirigée par les modèles.

En se basant sur des standards liés à la modélisation et aux transformations de modèles que nous allons présenter dans le chapitre suivant. Les chapitres 4 et 5 exposent nos travaux sur le développement d'un RTOS, en s'appuyant sur les concepts présentés dans les chapitres 1 et 3.

C H A P I T R E

3

Ingénierie Dirigée par les modèles

1. Introduction

Une nouvelle manière d'envisager la production et la maintenance des systèmes logiciels consiste à s'appuyer essentiellement sur les modèles, qui sont alors considérés comme entités de première classe. Il s'agit de l'ingénierie des modèles (ou IDM), qui est une branche de l'ingénierie des langages. Les modèles sont maintenant représentés à l'aide de formats précis dont la manipulation peut être automatisée. Chacun de ces modèles est défini en utilisant un langage spécifiant un ensemble de concepts et leurs relations. Chaque langage a généralement aussi une syntaxe concrète, par exemple textuelle ou visuelle, permettant de représenter les modèles.

L'IDM vise à définir un système logiciel à l'aide d'un ensemble de modèles utilisant différents langages. L'un des intérêts de l'IDM est de pouvoir considérer les modèles sur lesquels le programmeur raisonne comme faisant partie à part entière de la définition du logiciel.

L'IDM est connue sous différents noms. Une de ses appellations en langue anglaise est MDE (Model Driven Engineering pour ingénierie dirigée par les modèles). Les principes du MDE sont appliqués dans différents standards.

L'approche MDA (Model Driven Architecture ou architecture dirigée par les modèles) est un exemple d'application du MDE. Le MDA est recommandé par l'OMG (Objet Management Group) et est basé sur d'autres recommandations de ce même organisme. L'automatisation de la manipulation des modèles est donc réalisée par des opérations sur ces modèles.

2. L’architecture MDA de l’OMG

En novembre 2000, l’OMG annonçait son initiative MDA. Le consensus sur UML a été essentiel dans cette transition des techniques de production de logiciel basées sur le code vers des techniques de production basées sur les modèles. Un rôle clef est maintenant joué par le concept de méta-modèle. Mais ceci n’est pas suffisant.

Le MOF [MOF03] (Object Management Group, 1997) est issu de la reconnaissance qu’UML était un méta-modèle possible dans le domaine du développement logiciel, mais n’était pas le seul. Devant le danger de voir se développer et évoluer indépendamment une grande variété de méta-modèles différents et incompatibles (data warehouse, workflow, software process, etc.), il y

avait un besoin urgent de fournir un cadre global d'intégration pour tous les méta-modèles dans le domaine de l'ingénierie du logiciel, des systèmes et des données.

La réponse logique était donc d'offrir un langage de définition de méta-modèles, c'est-à-dire un méta-méta-modèle, chaque méta-modèle définissant lui-même un langage pour décrire un domaine spécifique d'intérêt. Par exemple UML permet de décrire les artefacts d'un logiciel à objets.

D'autres méta-modèles adressent des domaines différents comme le « legacy » (existant logiciel), les processus logiciels, l'organisation, les tests, la qualité de service, etc. Leur nombre est important et croissant. Ils sont définis comme des composants séparés et de nombreuses relations existent entre eux.

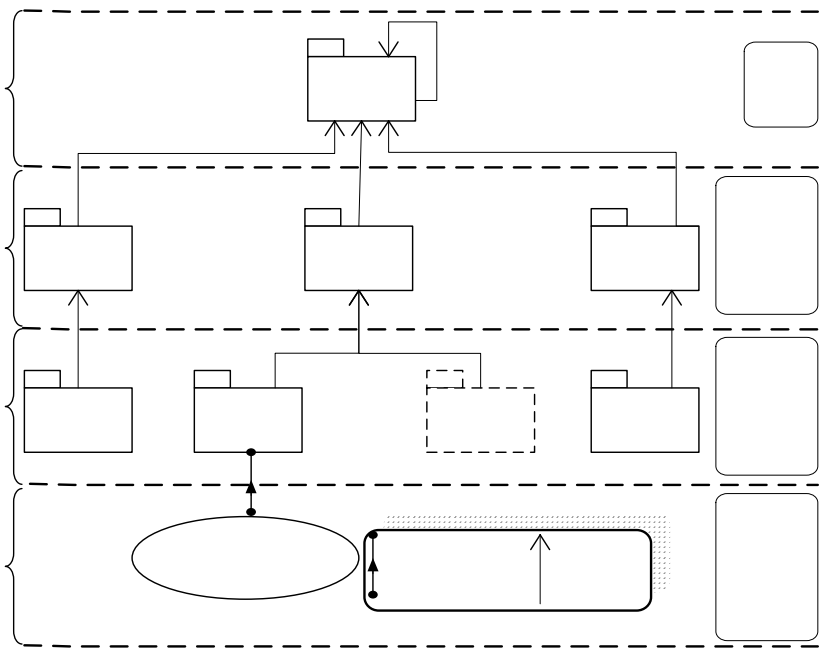

Figure 1 : *Interprétation de la pile de modélisation multi-niveau de l'OMG*

Le changement réel en ingénierie des modèles est intervenu lorsque ces modèles ont commencé à être utilisés directement dans les chaînes de production de logiciel. Bien que cette possibilité ait

Niveau
M3

souvent été considérée et partiellement appliquée, il est maintenant possible d'envisager son déploiement industriel à grande échelle (Greenfield et al., 2003).

Jusqu'à présent les modèles d'analyse et de conception ont principalement été utilisés pour documenter les systèmes logiciels. Les analystes et les concepteurs ont produit des modèles souvent fournis aux programmeurs comme du matériau d'inspiration, pour faciliter la production de logiciel. Le passage de cette période « contemplative » à une nouvelle situation où les outils de production seront dirigés par les modèles a été facilité par l'introduction de standards du MDA comme la recommandation XML [XML01] (Object Management Group, 1998).

La question de la transformation de modèles se situe aussi au centre de l'approche MDA **(figure** Une suggestion initiale de recherche avait initialement été faite en (Lemesle, 1998) et un appel à proposition industriel (RFP MOF/QVT) est actuellement en cours (Object Management Group, 2002) pour définir une sorte de langage unifié de transformation ou plutôt une famille coordonnée de tels langages. Ceci permettra de transformer un modèle Ma en un autre modèle Mb, indépendamment du fait que les méta-modèles MMa et MMb de Ma et Mb soient identiques ou différents. De plus le programme de transformation, par exemple écrit en langage ATL (Bézivin et al., 2004) - un langage de la famille QVT - doit lui-même être considéré comme un modèle Mt. En conséquence il sera conforme à un méta-modèle MMt, une définition abstraite de ce langage de transformation. Ces éléments constituent donc les briques de base de ce que l'on appelle l'architecture MDA de l'OMG. Ils évoluent rapidement, donnant lieu à la réalisation d'outils industriels applicables et parfois appliqués à certains domaines spécifiques.

Cependant, la communauté de recherche est elle-même impliquée dans la compréhension en profondeur des concepts et des principes régissant cette approche industrielle du MDA.

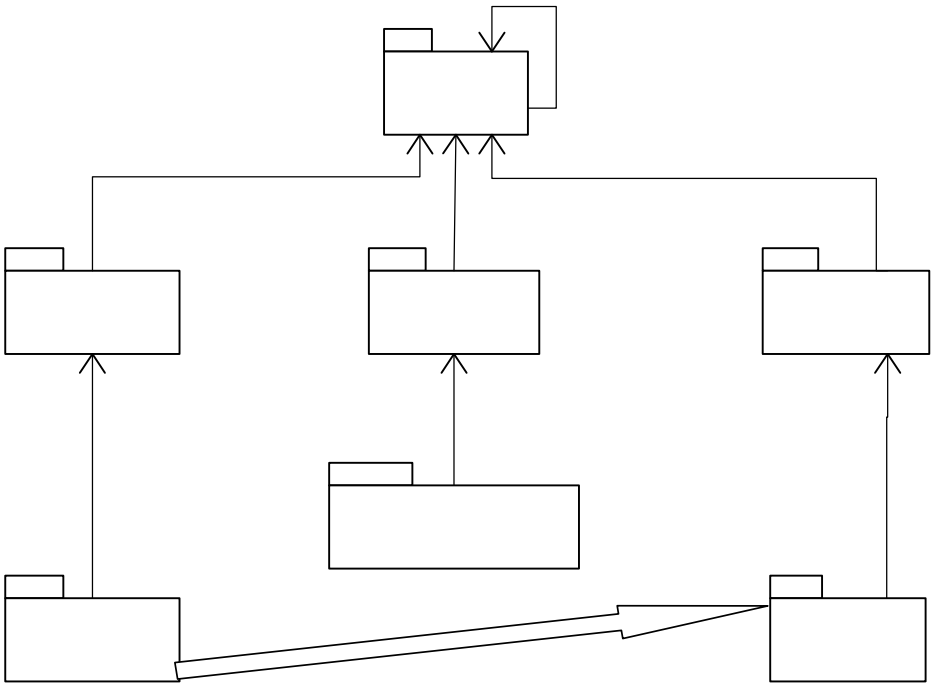

Figure 2: *La transformation de modèles basée sur les méta-modèles*

3. Principes de base

En technologie des objets, un principe de base (« Tout est objet », [P1]) a été très utile lors de l'apparition de la technologie sur la scène industrielle dans les années 1980 pour la pousser dans la direction de la simplicité, de la généralité et de la puissance d'intégration. De façon similaire, en ingénierie des modèles, le principe de base « Tout est modèle » [P2] possède plusieurs propriétés intéressantes.

Tout est objet	[P1]
Tout est modèle	[P2]

Tableau 3: *Principe [P1] et [P2]*

L'approche MDA n'est pas basée sur une idée unique. Parmi les objectifs poursuivis on peut mentionner la séparation de descriptions métier neutres d'avec les aspects liés à la plateforme,

l'expression séparée des aspects d'un système en cours de développement par des langages spécifiques de domaines, l'établissement de relations précises entre ces différents langages dans un cadre global et en particulier la possibilité d'exprimer des transformations précises entre eux, etc.

Comme suggéré par les figures 3 et 4, les outils conceptuels qui étaient les plus utilisés dans les années 1980 sont en cours de renouvellement. Au début de la période de déploiement industriel de la technologie des objets, ce qui était important était qu'un objet soit instance d'une classe et qu'une classe puisse hériter d'une autre classe. Ceci peut être vu comme une définition minimale correspondant au principe [P1]. Nous appelons ces deux relations instanceOf et inheritsFrom.

De façon très différente, ce qui semble être important aujourd'hui est qu'une vue particulière (un aspect) d'un système soit capturé par un modèle et que chaque modèle soit écrit dans le langage de son méta-modèle. Ceci peut être vu comme une définition minimale correspondant au principe [P2]. Nous appelons les deux relations de base representedBy et conformsTo.

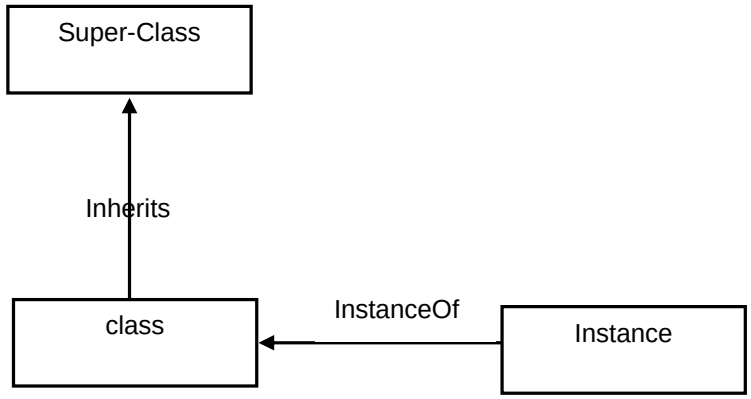

Figure 3 : *Notions de base en technologie des objets [P1]*

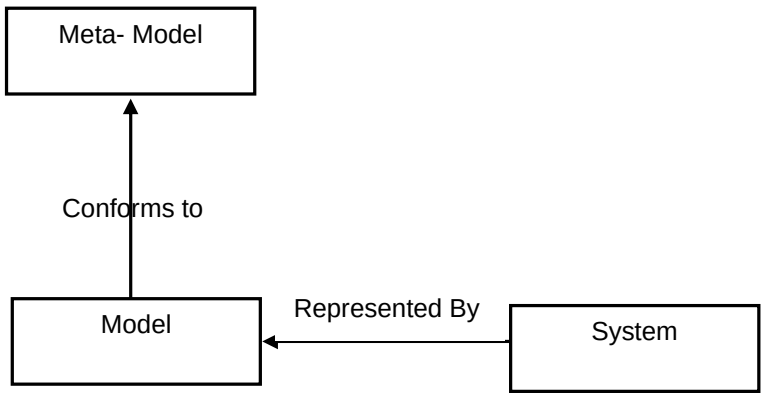

Figure 4: *Notions de base en ingénierie des modèles [P2]*

Lorsque l'on considère un système donné, on peut travailler avec différentes vues de ce système, chacune de celles-ci étant caractérisée de façon précise par un méta-modèle donné. Quand plusieurs modèles différents ont été extraits du même système à l'aide de méta-modèles différents, ces modèles restent liés et pourront être, recomposés par la suite. Pour que ceci puisse être largement appliqué, il est nécessaire de pouvoir disposer d'une organisation régulière des modèles composites. Il s'agit d'une organisation pragmatique similaire à l'organisation des langages de programmation déjà présentée dans la partie droite de **la figure 1.**

Récemment des sociétés comme Microsoft ou IBM ont défini leurs positions à ce sujet. Dans le « manifeste MDA » (Booch et al., 2004) d'IBM les principes de base sont au nombre de trois :

- **La représentation directe :** c'est-à-dire la nécessité de disposer de familles de langages de domaines (DSL) permettant de prendre en compte chacune des situations et des communautés corporatives
- **L'automatisation :** permettant les traitements de mise en correspondance automatique des modèles conformes aux différents langages de domaines
- **Les standards ouverts :** permettant l'émergence rapide d'écosystèmes d'industriels utilisateurs et fournisseurs d'outils et de chercheurs appliqués autour de plateformes de logiciel libre utilisant ces standards, par exemple ceux basés sur MOF ou sur XML.

4. Les standards MDA

Au cœur du MDA, se trouvent plusieurs standards importants : UML [UML01], XMI [XML01], MOF [MOF03] et le CWM [CWM01].

4.1. Les profils UML

Le MDA préconise fortement l'utilisation d'UML pour l'élaboration de PIMs et de la plupart des PSMs: les spécificités de chaque plate-forme peuvent être modélisées grâce aux mécanismes d'extension d'UML (stéréotypes, valeurs marquées, contraintes). En fait, on peut définir pour chaque système un profil qui regroupe les éléments nécessaires à ses caractéristiques. Nous voyons en détail la notion de profil au niveau du chapitre suivant.

Dans le cadre du MDA, les profils UML s'intègrent parfaitement car ils tirent partis des informations de sémantiques portées par le modèle PIM pour générer automatiquement les

classes et le framework associés au domaine cible. Outre l'interopérabilité plus aisée entre les applications tirant parties de ces frameworks, les profils UML permettent de gagner du temps de minimiser les risques lors de l'élaboration d'un logiciel.

4.2. MOF (Meta Object Facility)

Le langage MOF est le standard de l'OMG qui permettait l'élaboration de méta-modèles. Il est une extension du modèle objet, qui permet à celui-ci de représenter non plus des entités du monde réel, mais des entités de description de modèles. Le MOF est donc un langage de description des langages de modélisation (ou méta-modèles). En utilisant des éléments du MOF, on peut décrire un langage tel que UML, le langage de description des bases de données relationnelles, XML, ou encore le MOF lui-même. Ce langage permet d'abord de faire une représentation graphique d'un langage de modélisation particulier. Mais surtout il définit une grammaire selon laquelle un programme peut reconnaître et traiter des éléments d'un modèle décrit dans le langage spécifié.

Dans les diagrammes MOF, les classes (que l'on appelle des méta classes) représentent les concepts à définir et les associations (que l'on appelle des méta associations) représentent les relations entre ces concepts. Les méta classes et les méta associations sont contenues dans des packages. Un intérêt du MOF est qu'il permet de faire interopérer des méta-modèles différents. Une application MOF peut manipuler un modèle à l'aide d'opérations génériques sans connaissances du domaine.

4.3. XMI (XML Metadata Interchange)

XMI est le standard de l'OMG qui fait la jonction entre le monde des modèles et le monde XML de W3C (World Wide Web Consortium). XMI a été mis en place en 2000 pour permettre la sérialisation des modèles, afin de rendre possible l'échange de modèles entre différents logiciels (de modélisation, de développement). XMI offre ainsi une solution pour représenter des objets et leurs associations sous forme textuelle. De plus, puisque XMI est basé sur XML, les métadonnées (tags) et les instances (elements) sont regroupées dans le même document, ce qui permet à une application de comprendre les instances grâce à leurs métadonnées. De plus, la possibilité d'imbrication des balises permet de représenter l'imbrication entre éléments dans un méta-modèle, comme par exemple entre une classe et ses attributs.

Le standard XMI, est actuellement utilisé, sous de nombreuses variantes par des plateformes de modélisation et de développement logiciel, notamment Poseïdon, NetBeans et Eclipse, pour permettre la transmission de modèles entre ces plateformes.

Le standard XMI permet de décrire une instance du MOF sous forme textuelle, grâce au langage XML en définissant la manière d'utilisation de ses balises. Il permet la génération de DTDs (Document Type Definitions) et de schémas XML à partir de méta-modèles MOF. Les modèles sont alors traduits dans des documents XML conformes à leurs DTD correspondantes. Néanmoins, l'application la plus connue de XMI est celle qui a permis la construction de la DTD UML. Cette DTD UML permet la représentation des modèles UML sous forme de documents XML et assure ainsi les échanges de modèles UML entre les différents outils du marché. Le standard XMI de sérialisation des modèles compatibles à MOF est en cours de stabilisation et son utilisation devient incontournable dans les outils industriels.

4.4. CWM (Common Warehouse Metamodel):

Le CWM est le standard de l'OMG pour les techniques liées aux entrepôts de données. Il couvre le cycle de vie complet de modélisation, construction et gestion des entrepôts de données. Le CWM définit un méta-modèle qui représente les méta données aussi bien métiers que techniques qui sont le plus souvent trouvées dans les entrepôts de données. Il est utilisé à la base des échanges de méta données entre systèmes hétérogènes.

5. Modélisation/Méta-modélisation

Un modèle représente un système réel en se basant sur la sémantique et les règles qui conditionnent ses éléments; en d'autres termes il ne doit en aucun cas briser la structure ou les contraintes que les éléments du système réel respectent.

En conséquence, les éléments du langage d'expression d'un modèle doivent satisfaire un ensemble de règles qui leur permet de former un modèle qu'on appelle méta-modèle. Le langage est alors dit un langage bien défini et on peut ne plus faire la distinction entre le méta-modèle et le langage qu'il définit (voir **figure 5**)

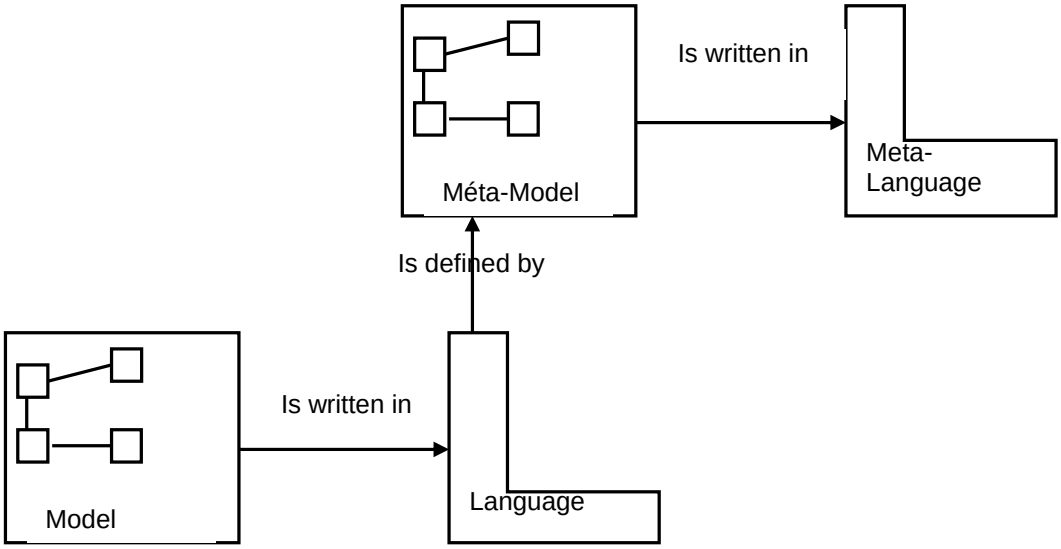

***Figure 5** : Modèles, langages, méta-modèles et métalangages*

De même, un méta-modèle est écrit dans un langage appelé métalangage et il est instance d'un métaméta-modèle qui définit les éléments du métalangage.

On peut même dire qu'un langage est écrit (défini) par un métalangage. Le métalangage doit être écrit encore par un méta métalangage et ainsi de suite.

La figure 6 illustre les 4 niveaux de modélisation :

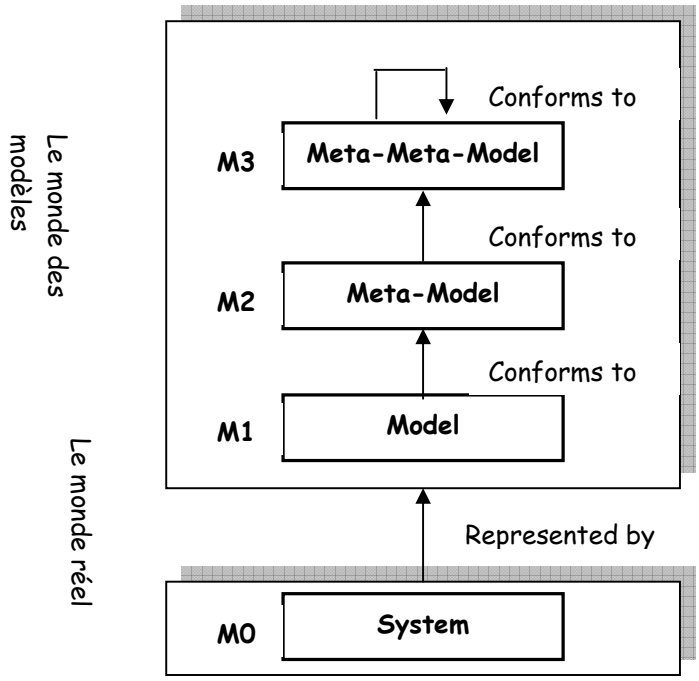

***Figure 6 :** L'organisation 3+1 du MDA*

On distingue alors selon la normalisation de l'OMG quatre couches de modélisation :

- ***m*0 :**C'est la première couche correspondant à un système en exécution (*running*). *m*0 contient des instances d'objets en cours de traitement par le logiciel.
- ***m*1 :**C'est la couche du modèle exécutable renfermant la structure et le comportement du système. C'est à ce niveau que se situent généralement les modèles que nous manipulons quotidiennement dans nos activités de développement de logiciel.
- ***m*2 :** C'est la troisième couche de modèles appelés méta-modèle, et dont les instances sont des modèles de la couche *m*1.
- ***m*3 :** C'est un niveau d'abstraction encore plus élevé, où l'instance d'un modèle de cette couche donne un modèle de la couche *m*2. Un modèle de *m*3 donnera une syntaxe d'écriture de méta-modèle.

OMG arrête cette suite d'abstraction au niveau 4 en définissant les éléments de *m*3 comme instances de concepts de la même couche *m3*. C'est à dire que *m*3 est définie d'une façon auto descriptive.

On rappelle que le méta-modèle est conforme à lui-même.

5.1. Langages de méta-modélisation :

Il existe de nombreux langages de méta-modélisation tels que MOF 1.4, EMOF 2.0 [KMM05], Ecore, etc. La plupart de ces langages utilisent les concepts de classe, attribut et association ou référence. Cependant, ils sont incompatibles puisqu'un méta-modèle conforme à l'un d'entre eux ne peut généralement pas être conforme à un autre. Ils n'ont pas non plus de définition formelle. De plus, certains de ces langages possèdent des concepts dépassant le domaine de la méta-modélisation. C'est notamment le cas de Ecore qui permet par exemple d'annoter un méta-modèle avec des directives pour la génération de code Java. Par ailleurs, il n'existe que peu d'outils pour la création ou la manipulation des méta-modèles définis dans ces langages.

On s'intéresse dans notre mémoire au langage de méta-modélisation KM3 [KMM05] (Kernel MetaMetaModel ou métaméta-modèle noyau) qui a une définition formelle basée sur la logique du premier ordre. Ce langage est une simplification des langages existants dans ce domaine et

plus particulièrement de MOF 1.4, EMOF 2.0 et Ecore. Il est donc possible de traduire n'importe quel méta-modèle KM3 vers un de ces langages. Les méta-modèles définis en KM3 peuvent ainsi être utilisés avec différents systèmes de manipulation de modèles basés sur ces métaméta-modèle (e.g. Netbeans/MDR et Eclipse/EMF). Par ailleurs, afin d'offrir une alternative à l'utilisation des quelques éditeurs graphiques de méta-modèles, une syntaxe concrète textuelle a été définie pour KM3.

L'objectif de KM3 est de fournir une solution relativement simple à la définition du méta-modèle de définition du domaine (ou DDMM pour Domain Definition MetaModel) d'un DSL. KM3 est donc un DSL pour la définition des méta-modèles tout comme EBNF est un DSL de définition de grammaires :

- **Méta-modèle de définition du domaine :** Le DDMM de KM3 est un métaméta-modèle auquel les autres DDMMs sont conformes. Ce DDMM est défini en KM3, de la même manière que la syntaxe EBNF peut être décrite en EBNF en quelques lignes.

KM3 utilise les concepts de classe (Class), attribut (Attribute), référence (Reference), etc. Sa structure est proche d'EMOF 2.0 et d'Ecore.

- **Syntaxe concrète :** La syntaxe par défaut de KM3 est textuelle .Ceci permet la définition simple de méta-modèles avec n'importe quel éditeur textuel.
- **Sémantique :** La sémantique de KM3 permet la définition de méta-modèles et de modèles. Une définition conceptuelle précise de KM3 est présentée ci après.

Des transformations vers et depuis MOF 1.4 et Ecore ont notamment été définies en ATL [BEZ03]. KM3 est de ce fait utilisable avec des outils tels qu'Eclipse EMF et Netbeans MDR.

En tant que métaméta-modèle, KM3 est plus simple que MOF 1.4, MOF 2.0 ou encore Ecore. Il ne contient que 14 classes là où Ecore en a 18 et MOF 1.4 en a 28. Seuls les concepts essentiels de ces autres métaméta-modèles ont été retenus dans KM3.

5.2. Transformation des modèles

La principale opération dans MDA est la transformation de modèles qui consiste à créer de nouveaux modèles à partir de modèles existants. La transformation n'est alors plus limitée à la traduction du code source mais peut opérer sur tous les modèles décrivant un système. De plus,

les modèles créés par transformation ne sont pas nécessairement du code exécutable mais peuvent être très variés. Dans son approche MDA, l'OMG recommande QVT [MOF03] (Query / View / Transformation ou requête / vue / transformation) qui propose une famille de langages de transformation.

5.2.1. Caractéristiques :

La transformation de modèles est formée d'un ensemble de règles et doit être écrite dans un langage bien défini qu'un outil pourra compiler et exécuter. Cette automatisation sous-entend que les modèles doivent être écrits dans un langage traitable par les machines. La figure 7 illustre ceci tout en utilisant la notion de langage d'écriture de modèle qu'on a examiné dans la section II.

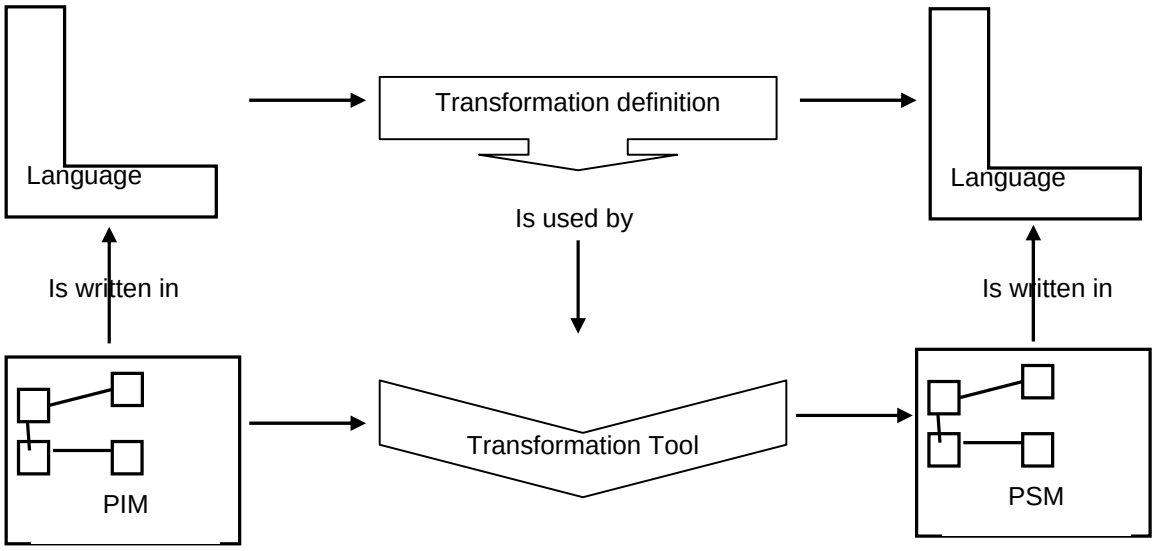

Figure 7 : *Automatisation des transformations*

Une transformation se caractérise par la manière dont ses règles s'appliquent, d'où le besoin de distinguer les propriétés suivantes qui varient selon la définition de la transformation et le langage utilisé :

- L'ordre d'application sur les fragments de modèle correspondant à une règle;
- L'ordre dans lequel s'appliquent les différentes règles;
- La possibilité de composition de règles;
- La relation entre les modèles source et cible (même, différent);
- La traçabilité (définie ci-après);

- La bidirectionnalité;
- L'interactivité permettant la paramétrisation ;

5.2.2. Langages de transformation

Pour définir une transformation de modèles, on peut utiliser un langage non formel, un

langage d'action pour représenter l'algorithme de la transformation, ou encore un langage bien défini de mapping de modèles.

Le QVT RFP a déjà généré 8 propositions ainsi qu'un certain nombre d'initiatives indépendantes. Nous allons citer quelques-unes rapidement en insistant sur les langages s'avérant utiles à notre future contribution dans le domaine. De plus amples informations sur les outils d'implémentation seront données ultérieurement.

- **OpenQVT [QVT03] :** soumis par un groupe d'universités et de sociétés françaises en réponse au QVT RFP. Ce langage est à la base du développement de l'implémentation de la syntaxe ATL par un outil prometteur de transformation de modèles ADT qui est intégrable dans l'environnement de développement Eclipse
- **XMOF d'IBM et Compuware :** Comparable à XSLT qui est un langage applicable pour exprimer des transformations de fichiers XML, XMOF est un langage déclaratif qui décrit les résultats voulus d'une transformation plutôt que les processus nécessaires à sa réalisation.
- **Le langage TRL :** Ce langage est une autre proposition en réponse au QVT RFP soumise par Alcatel Softeam, Thales, TNI-Valiosys et Codagen Technologies Corp, et supportée par France Telecom, INRIA/IRISA, Softeam, Université de Paris VI, Université de Nantes, LIFL, CEA.,TRL utilise les standards de l'OMG comme OCL, pour exprimer les contraintes et les *queries* de MOF. De plus, ce langage est basé sur une technique de méta-modélisation avec les deux approches déclarative et impérative. Il permet le marquage et la paramétrisation, et il accepte plus qu'un modèle d'entrée à la fois.
- **QVT Partners:** Est une soumission de proposition QVT qui se distingue par la considération de l'aspect unidirectionnel et bidirectionnel des transformations. Ses LHS et

RHS ressemblent à ceux d'UMLX, qu'on verra dans cette section, mais sans supporter les multiplicités. De plus ce langage compte sur l'expression textuelle de la relation entre LHS et RHS qui forme une contrainte qui doit être satisfaite après tout passage de LHS vers RHS. Ce langage utilise Action Semantics pour l'expression des *mappings* en plus d'OCL.

- **MOLA :** Le laboratoire IMCS de l'université de Latvia a développé le Model Transformation Language (MOLA) comme une tentative de définition de transformations plus **naturelles** et **lisibles**. On a adopté des structures de contrôle itératives et simples plutôt que récursives puisées dans la programmation structurelle traditionnelle.,MOLA utilise l'approche de réécriture graphique utilisant les LHS et RHS avec une instruction essentielle 'loop' qui cherche tous les fragments de graphes coïncidant avec un LHS.
- **ATL** [ATL05] **:** L'équipe ATLAS a proposé à la normalisation de l'OMG un langage de transformation de modèle nommé ATL (ATLAS Transformation Language) qui sera détaillé dans la section suivante.

5.2.3. Le langage ATL

ATL est un langage de transformation hybride, il contient un mélange de constructions déclaratives et impératives, cependant, l'utilisation du style déclaratif est encouragée. Les transformations ATL sont unidirectionnelles et opèrent sur des modèles source en lecture seule et produisent des modèles cible en écriture seule Une transformation bidirectionnelle est typiquement implémentée par un couple de transformations : une pour chaque direction.

Pendant l'exécution d'une transformation, les modèles source peuvent être navigués mais pas modifiés, alors que les modèles cible ne peuvent pas être navigués. Ces restrictions permettent de simplifier la sémantique d'exécution et notamment de garantir un résultat déterministe sans demander au développeur de définir explicitement un ordre d'exécution des règles.

La structure d'ATL est comme suit :

- **L'en-tête :** commence par le mot-clé module suivi du nom du module. Ensuite, les modèles source et cible sont déclarés comme des variables typées par leurs méta-modèles. Le mot-clé create indique les modèles cible. Le mot-clé from indique les modèles source.

Le modèle cible est représenté par le variable **OUT** à partir du modèle source représenté par **IN**.

```
module <nom du module> ;
create OUT : <métamodèlecible> from IN : <métamodèle source>;
```

Figure 8 : *Syntaxe de l'en-tête*

- **Helpers :** Les fonctions ATL sont appelées helpers d'après le standard OCL sur lequel ATL se base. En ATL, un helper peut être spécifié dans le contexte d'un type OCL (par exemple String ou Integer) ou d'un type source (venant de l'un des méta-modèles source). OCL définit deux sortes de helpers : opération et attribut.

✓ **Les helpers opération :** peuvent être utilisés pour définir des opérations dans le contexte d'un élément de modèle ou du module de transformation.

Le rôle principal des helpers opération est de réaliser la navigation des modèles source. Ils peuvent avoir des paramètres et peuvent utiliser la récursivité. Les helpers opération définis dans le contexte d'éléments de modèles permettent les appels polymorphiques.

Puisque la navigation n'est autorisée que sur les modèles source en lecture seule, une opération retourne toujours la même valeur pour un contexte et un ensemble d'arguments donnés.

✓ **Les helpers attribut :** sont utilisés pour associer des valeurs nommées en lecture seule sur les éléments de modèles source.

Comme les opérations, ils ont un nom, un contexte et un type. La différence est qu'ils ne peuvent pas avoir de paramètres. Leur valeur est définie par une expression OCL.

Comme les opérations, les attributs peuvent être définis récursivement avec les mêmes contraintes de terminaison et de cycles.

Les helpers attribut sont pratiquement comme les propriétés dérivées de MOF 1.4 ou d'Ecore mais peuvent être associés à une transformation. Ils ne sont pas nécessairement liés à un méta-modèle donné. Alors que dans EMF et MDR ils sont implémentés en Java, ils sont définis en OCL avec ATL.

Les helpers attribut peuvent être considérés comme un moyen de décorer les modèles source avant l'exécution de la transformation. La décoration d'un élément de modèle peut dépendre de la déclaration d'autres éléments. De plus ils peuvent aussi être utilisés pour établir des liens entre éléments de différents modèles source.

Le type d'un attribut peut en effet être une classe provenant d'un méta-modèle différent du méta-modèle de son contexte. Ceci correspond à une forme élémentaire de composition de modèle.

La figure 3 donne la forme des helpers ATL :

```
Helper def : (<paramètre> : <type>) : <type retour> =
<expression OCL>;
```

Figure 9 : *Forme d'une méthode dans ATL*

- **Règles de transformation :** La règle de transformation est la construction élémentaire en ATL pour exprimer la logique de transformation. Les règles ATL peuvent être soit déclaratives soit impératives.

✓ **Règles déclaratives (matched rules) :** Une matched rule est composée d'un pattern (motif) source et d'un pattern (motif) cible, sa syntaxe est la suivante :

```
rule <nom de la règle> {
 from
<pattern source>
 to
<un ou plusieurs patterns cibles>}
```

Figure 10 *: Matched rules*

✓ **Règles impératives :** on distingue :

- **Called rules :** se sont des opérations au sens OCL et peuvent donc être appelé comme des fonctions dans les expressions. Ils ont des paramètres.
- **Action block :** se sont des séquences d'instructions impératives qui peuvent être utilisées soit dans matched ou called rules.

6. Conclusion

Nous avons étudié dans ce chapitre l'approche MDA qui base entièrement le processus de développement sur le concept des modèles à l'opposé des approches traditionnelles d'analyse/codage. La logique métier est conçue uniquement de manière abstraite, indépendante de toute technologie d'implémentation. Les différentes étapes de transformation et d'enrichissement apportant des propriétés non fonctionnelles au modèle de base amènent, de manière plus ou moins automatisée, à une application exécutable dans un environnement choisi. Cette méthode nous apparaît comme particulièrement intéressante dans nos travaux sur le développement d'applications à contraintes de temps réel en environnements embarqués. Elle sera développée utilisée tout au long des deux chapitres suivants, notamment pour répondre aux objectifs que nous nous sommes fixés. Ainsi nous étudierons, d'une part, comment la modélisation indépendante de la plate-forme peut servir de base pour la spécification de contraintes non fonctionnelles essentiellement celles temps réel, également indépendante de l'environnement cible. Puis nous présenterons, d'autre part, les processus de transformations, permettant de passer du niveau indépendant de la plate-forme vers un niveau plus spécifique, qui seront utilisés pour générer le code applicatif et le code en tenant compte de paramètres spécifiques à un environnement donné.

C H A P I T R E

4

Démarche de conception proposée

1. Introduction

Ce chapitre présente l'objectif de ce Mastère. Nous avons présenté dans le premier chapitre les STRE aux quels nous nous intéressons, en nous focalisant sur l'étude des RTOS. Dans le second chapitre, nous avons synthétisé les principales approches de modélisation de ces systèmes. Il ressort de cette synthèse une volonté d'intégration de l'RTOS lors de la modélisation d'un STRE, ainsi que la définition explicite de la sémantique temporelle et transitionnelle dans le but d'assurer le bon fonctionnement du système et de valider son ordonnançabilité. Il s'avère en réalité que les gains obtenus sont doubles : gain en terme de conception, de réutilisation, de maintenance; et gain en termes de qualité à l'exécution. Nous sommes donc confortés dans l'idée d'adopter la démarche MDA, pour étudier la modélisation de l'RTOS.

Ce chapitre présente alors la contribution liée à la modélisation orientée objet de l'RTOS, en s'appuyant sur le niveau le plus haut des modèles MDA, et notamment en focalisant l'étude sur le niveau PIM, plus précisément en nous intéressons au modèle source lors de la phase de transformation de modèles.

2. Identification des besoins

Dans cette section, nous présentons notre étude sur la modélisation de l'RTOS en se basant sur l'approche dirigée par les modèles et sur les travaux de l'OMG. Comme nous l'avons décrit dans le troisième chapitre, MDA est une approche dirigée par les modèles qui propose des spécifications pour la création, la visualisation et l'échange de modèle logiciels. Deux niveaux de modèles coexistent : un niveau abstrait et indépendant de l'environnement d'exécution, nommé PIM et un niveau dépendant du support d'exécution, nommé PSM. Cette section s'attache à la description abstraite des RTOS, c'est-à-dire elle est attachée au premier niveau.

Dans le contexte de notre travail, les profils UML existants, décrits dans le deuxième chapitre, s'intéressent uniquement à la description de l'architecture et de l'application d'un STRE. Ils ne supportent pas l'intégration d'une composante qui devient de plus en plus indispensable dans les nouveaux STRE à savoir l'RTOS.

Le point fort de notre travail vient contredire la multiplicité des environnements et renforcer l'orientation d'unification dans le but d'avoir un profil unifié. Pour cela, nous avons essayé de

partir des profils existants, de prendre en compte les exécutifs temps réel, et de définir des sémantiques temporelles et transitionnelles.

Pour cela, les profils déjà identifiés vont nous apporter une base pour construire un profil UML orienté RTOS, et pour lequel nous nous sommes fixés les contraintes suivantes :

- Créer un modèle d'un système d'exploitation temps réel, correspondant au niveau PIM de l'approche MDA afin de prendre en considération les caractéristiques communes d'un RTOS et les services génériques qu'il offre.
- Garder à l'esprit le critère de l'indépendance vis-à-vis de la plate forme d'exécution.
- Restreindre le modèle d'ordonnancement aux cas monoprocesseur. La possibilité de l'extension vers le multi processeurs reste valable.
- Définir explicitement toutes les contraintes temporelles du système et mettre l'accent sur l'aspect déterminisme du système, il s'agit de définir un modèle d'ordonnancement.
- Assurer la cohérence entre la vue statique et la vue dynamique des modèles: En effet, la construction d'un modèle d'application repose d'abord sur la vue statique, notamment par le biais du diagramme de classes. Les informations liées à l'RTOS décrites dans celui-ci servent donc de base par défaut pour l'intégration de l'exécutif temps réel lors de la modélisation d'un STRE. Les informations d'interactions décrites par le biais du diagramme d'états transitions viennent ensuite se greffer pour compléter, spécialiser et ordonnancer les tâches décrites dans le diagramme de classes.

Nous proposons alors un modèle qui vient complémenter et enrichir ce qui existe, ce modèle doit spécifier les contraintes d'exécution, en s'appuyant sur le vocabulaire habituellement employé dans les systèmes tels que l'allocation de ressources, les politiques d'ordonnancement, les délais d'invocation d'opérations, etc). En résumé, nous proposons une démarche qui part de la modélisation de la structure de l'RTOS jusqu'à la genèse du modèle d'ordonnancement. Une fois le modèle d'ordonnancement est obtenu, nous passons à la génération de code.

3. Démarche proposée

Certes, le diagramme de classe présente une vue statique d'un système, mais ne peut pas couvrir tout l'aspect comportemental du système. La séparation explicite lors de la modélisation de la

structure de l'RTOS et de l'ordonnanceur et le manque de cohérence entre les différents diagrammes utilisés pourraient influer sur l'aspect comportemental du système.

Notre approche présente une démarche assurant la cohérence entre les différents diagrammes UML utilisés et couvrant l'aspect comportemental du système ainsi que ses contraintes temps réel. Il s'agit de modéliser, dans un premier temps la structure de l'RTOS. Dans un deuxième temps, nous associons un digramme statecharts relatif à l'entité tâche la plus importante dans le modèle et ce tout en l'annotant avec des contraintes OCL. Nous définissons ensuite les variations sémantiques temporelles et transitionnelles présenté par les statecharts. Ce qui nous amène à l'implémentation d'un profil UML pour l'implantation des statecharts. Cette phase sera bien décrite dans le chapitre suivant. Lors de la définition des variations sémantiques, nous appliquons quelques techniques à savoir la réification et l'énumération des états et des événements. Nous optons pour l'utilisation d'une démarche d'intégration de design pattern dans le but de réutiliser des composants logiciels existants et éprouvés, plutôt que de recréer de nouveaux modèles pour l'implémentation des statecharts. Le modèle final relatif à la variation des différents états relatifs aux comportements d'une tâche correspond au modèle cible lors de l'étape de transformation des modèles. Comme étape finale, nous passons à la génération automatique de code.

Notre démarche illustrée par la figure 11, pourrait alors se résumer aux étapes suivantes :

1. Définition de la structure de l'RTOS à l'aide d'un diagramme de classe (définition du modèle source)
2. Définition des différents états possibles pour la description du comportement d'une tâche à l'aide des statecharts
3. Définition des variations sémantiques associées aux statecharts
4. Implantation des statecharts
5. Construction du modèle cible correspondant au modèle cible du niveau PIM de la démarche MDA. Ce modèle est issu de la phase précédente
6. Génération de code

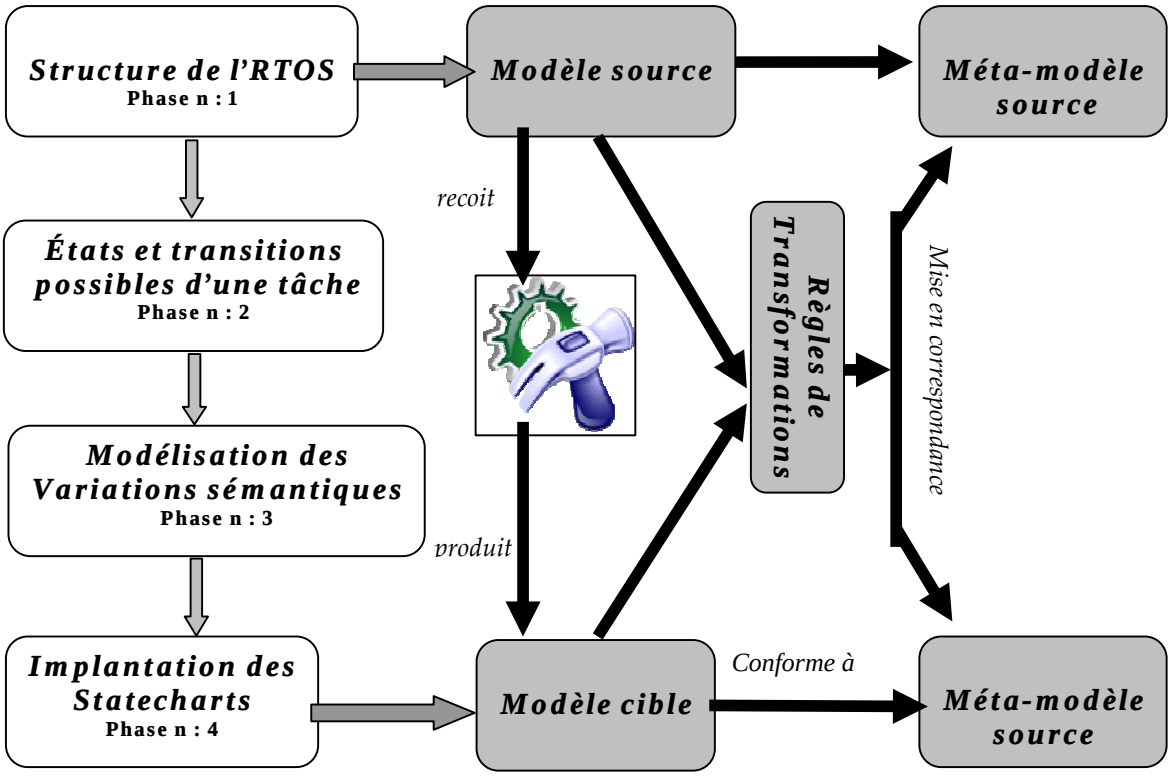

Figure 11 : *Démarche proposée*

Nous associons au cours de la deuxième phase de notre proposition, un diagramme d'états transitions décrivant une vue comportementale de l'état d'une tâche. Nous exprimons quelques règles OCL relatives à chaque état dans le but d'atteindre la qualité correction du comportement du système.

La suite de la description des autres phases de notre démarche sera décrite en détail dans le chapitre suivant.

4. Modélisation de la structure d'un RTOS

Un modèle d'RTOS peut être vu selon deux composantes : la structure et le modèle d'ordonnancement. A chaque modèle, nous associons les diagrammes convenables à savoir les diagrammes à aspect statique et/ou dynamique. A ce stade, les digrammes proposés doivent être cohérents et doivent couvrir touts les constituants d'un RTOS essentiellement ceux qui le caractérisent par rapport à un système d'exploitation ordinaire. Pour assurer l'extensibilité des profils, le modèle doit préserver leurs sémantiques et y rajouter les notions dont on a besoin.

Pour y arriver, nous nous basons sur deux travaux : Le premier correspond à la modélisation d'un RTOS dédié à l'automobile appelé OSEK [SHO04] et le deuxième concerne le système d'exploitation VxWorks [DAV06], il consiste à faire de la «reverse engineering».

Dans [SHO04], les auteurs se basent sur deux diagrammes de classe indépendants : un diagramme décrivant la structure et un autre décrivant l'ordonnanceur. Ces modèles séparés explicitement souffrent d'un manque de cohérence et de définition de sémantiques temporelles. En fait, le diagramme utilisé pour la caractérisation de l'ordonnanceur et un diagramme statique de façon qu'il ne puisse pas couvrir le comportement temporel d'un RTOS, il doit aussi posséder l'aptitude d'être complémentaire au modèle de la structure et ce via la bonne expression du suivi de l'évolution d'un processus temps réel.

Partant d'une bibliothèque temps réel écrite en langage C, DAV et Al [DAV06] effectuent les transformations nécessaires pour aboutir à un diagramme UML. Cette transformation amène à des entités spécifiant des composantes d'un système temps réel dont le lien entre elles est laissé à la charge du designer. Elle est restreinte à une description statique quoique le comportement puisse être touché en introduisant des attributs décrivant l'état de progression d'une tâche dans le temps ou en définissant une relation réflexive de précédence, cette technique s'appelle la définition de sémantique opérationnelle. Il est à noter que nous pouvons modéliser un algorithme d'ordonnancement via l'utilisation d'un diagramme de séquence mais cette solution reste inefficace vu la multiplicité des algorithmes d'ordonnancement et la difficulté de leurs intégrations à une démarche MDA.

Pour nos modèles proposés, nous décrivons la structure de l'RTOS à travers un diagramme de classe qui inclut la définition de la sémantique opérationnelle pour présenter la structure de l'RTOS. Nous définissons ensuite le comportement d'une tâche qui constitue le noyau d'un exécutif temps réel, afin d'assurer la cohérence entre les différents diagrammes UML. Pour aboutir au modèle d'ordonnancement, nous définissons les variations sémantiques temporelles et transitionnelles relatives aux différents états d'un processus temps réel.

Dans la suite de cette section, nous présentons les différentes techniques qui peuvent être utilisé pour modéliser un RTOS à savoir les modèles statiques et les modèles dynamiques. Nous justifions au fur et à mesure nos modèles proposés.

4.1. Modèle statique

Ce modèle correspond à la définition des composantes les plus essentielles d'un RTOS, toutefois il ne peut pas couvrir l'aspect comportemental de l'RTOS et plus particulièrement l'aspect déterminisme du système.

La description de la structure de l'RTOS peut être inspiré de la spécification de l'RTOS OSEK [SHO04] utilisé dans le domaine de l'automobile. Le diagramme utilisé est celui de classe présenté par la figure ci-dessous :

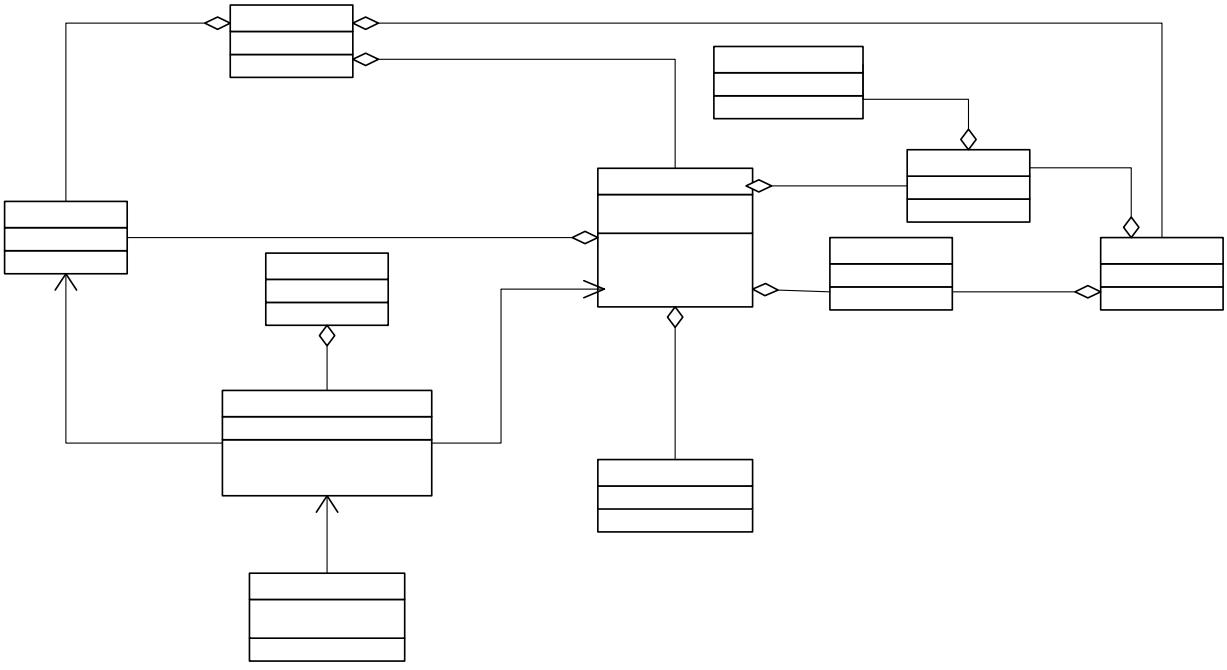

Figure 12*: *Diagramme de classe correspondant à la structure de l'RTOS OSEK

D'après la figure 12, la structure de l'RTOS est définie par les entités suivantes :

- **Task:** C'est le composant le plus important du noyau d'un RTOS. Ce dernier doit acquérir un grand nombre d'informations concernant les tâches dans le but de bien gérer

leurs ordonnancements ainsi que leurs exécutions. Cette mission est confiée à l'Ordonnanceur.

- **Event :** Au moment du déclenchement d'un évènement, l'état d'une tâche est changé.
- **ISR** *:* Interrupt Server Routine: C'est la routine chargée du traitement de l'interruption. Elle fait, dans ce contexte, le relais entre le mécanisme matériel d'interruption et le mécanisme logiciel de signalisation. Le concept d'interruptions est basé sur différentes catégories d'ISR, le premier (ISR type 1) est celui dans lequel on n'a pas besoin d'appeler un service exécutif. L'exécution du code de l'ISR est donc transparente pour l'exécutif et se traduit par un retard temporel de la tâche interrompue, durant le service de l'interruption, le deuxième modèle (ISR type 2) est celui dans lequel la routine est déclarée comme une routine d'interruption par un mot clé spécifique (le générateur d'application génère alors les appels de service nécessaires pour signaler à l'exécutif l'entrée et la sortie d'une routine d'interruption).
- **Message** : Il est utilisé pour échanger des messages de données entre un système expéditeur et un autre récepteur. Ce message est utilisé pour assurer l'envoi d'informations protégées entre les différentes tâches.
- **Alarm :** Basée sur un compteur, une alarme pourrait activer une tâche, imposer un évènement ou activer un alarmCallBack, elle est définie alors par les attributs suivants *:* activateTask, setEvent et activateAlarmCallBack. Une alarme peut être unique ou cyclique, absolue ou relative. Si elle est relative, la valeur spécifiée par un paramètre du service est un incrément par rapport à la valeur courante du compteur (expression d'un délai de garde par exemple) ; si elle est absolue, la valeur spécifiée par un paramètre du service définit la valeur du compteur qui active l'alarme. Une autre valeur est spécifiée dans le cas d'une alarme cyclique afin de préciser (en nombre de ticks) la valeur du cycle. Ainsi on sait simplement, sur un compteur lié à l'horloge temps réel, définir au travers de plusieurs alarmes, des tâches périodiques de périodes différentes.
- **Counter :** Un compteur présente une source logicielle/matérielle pour une alarme**,** il est présenté par les deux paramètres suivants *:* maxAllowedValue et minCycle. C'est un objet destiné à l'enregistrement de « ticks » en provenance d'une horloge ou d'un dispositif quelconque émettant des stimulis. C'est un dispositif de comptage ayant une certaine

dynamique, qui repasse à zéro après avoir atteint sa valeur maximale (valeur définie à la génération de l'application). Il compte les ticks après une éventuelle pré-division (par exemple 10 ticks représentent une unité pour le compteur). Plusieurs alarmes peuvent être associées à un même compteur, ce qui permet de constituer facilement, par exemple, des bases de temps.

- **Ressource** : Cette entité est utilisée pour cordonner les accès concurrents aux ressources partagées par les tâches. L'utilisation de cet objet est similaire à l'utilisation des sémaphores.

Une autre façon de modéliser consiste à procéder par une approche ascendante, il s'agit de la « reverse engineering ». Dans cette approche, Dav et Al [DAV06] partent d'une libraire écrite dans un langage de haut niveau (langage C) qui inclut une gamme de services formés par l'RTOS VxWorks et dégage le modèle adéquat. Ce dernier permet d'utiliser d'une manière simplifiée les mécanismes offerts par un système temps réel : tâche, événement (utilisant un sémaphore binaire), sémaphore d'exclusion mutuelle, activations périodiques et gestion du temps.

Pour répondre aux objectifs actifs d'un diagramme de classe UML, il est nécessaire de créer des tâches périodiques ou tâches apériodiques (déclencheurs d'événements), ce qui permettent les classes de base Processus, ProcessusPériodique, et ProcessusDeclenche (voir figure 13). Chaque objet actif hérite de l'une de ces classes selon le type 'activation de l'objet. A l'instanciation de l'objet, une tâche sera lancée, exécutant une méthode virtuelle de la sous-classe correspondante à l'objet actif.

A fin de gérer les flots de données entre objets actifs et au lieu de protéger l'accès aux méthodes des objets actifs, un objet partagé ayant une interface générique, la classe ElementCommunication a été mise en place. Deux méthodes, lire() et ecrire() permettent de mettre à jour ou de récupérer la valeur de cette variable partagée. Ensuite, selon les besoins, le concepteur parmi les objets de son exécutifs. Par exemple, la classe variableProtgee, qui implémente cette interface, associe un mécanisme de protection de données (sémaphore d'exclusion mutuelle) à travers ces méthodes, garantissant qu'un processus ne pourra passer outre et la classe BoiteAuxLettres utilise une file de message.

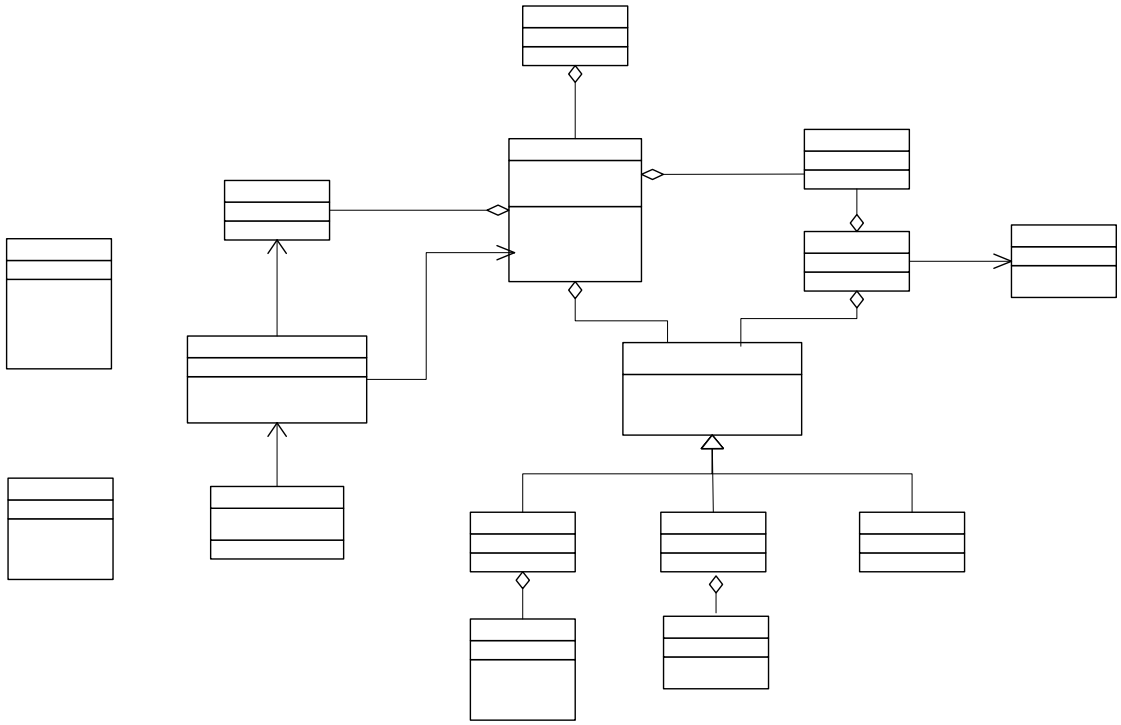

Figure 13*: *Bibliothèque d'objets VxWorks

La modèle présenté par la figure 13 souffre d'un manque de relation et de cohérence entre les entités, ceci est du à la difficulté du passage d'un code écrit en langage fonctionnel C à un modèle orienté objet. Ce qui nous a laissé penser à ajouter des relations et des associations entre certaines entités séparées tels que « Horloge », « WatchDog », « Evnet » et « MeanOfCommunication » .

4.2. Modèle dynamique

Suite à la description statique de la structure de l'RTOS, une définition de l'aspect comportemental du système s'avère très importante. Un modèle illustrant l'évolution d'une seule tâche dans le temps est nécessaire pour lui donner l'aspect déterminisme. Ce modèle serait ne supporte pas plusieurs tâches, il est différent du modèle d'ordonnancement. Ceci peut être modélisé en définissant le comportement d'une tâche ou encore en exprimant la dépendance d'une tâche avec une autre ou en indiquant le taux de progression d'une tâche dans le temps.

4.2.1. Modélisation du comportement d'une tâche

Etant donné que les machines d'états finies ou encore dit FSM (Final State Machine) présentent, par rapport à une entité, ses états possibles et ses transitions qui le font évoluer et spécifier ce que doit faire l'objet en réponse aux événements (ou traitements) qui lui sont appliqués, une FSM est attachée dans [SHO04] à l'entité Task, elle présente les états possible que peut subir une tâche suite au déclenchement d'un événement.

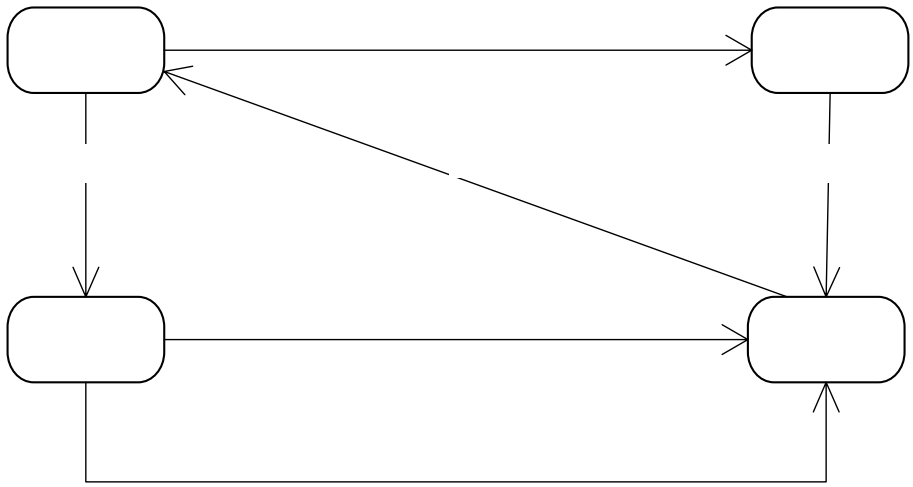

Figure 14**:** *Conception de la variation de l'état d'une tâche à l'aide des FSM*

La figure 14 illustre l'évolution détaillée de l'état d'une tâche d'un RTOS multi tâches, les valeurs qui lui sont associées sont présentées par les états suivants :

- Ready : En attente de l'acheminement d'une autre tâche
- Suspended : Soit la tâche est terminée ou elle est arrêtée par l'Ordonnanceur
- Waiting : En attente de la réalisation d'un événement
- Running : en cours d'exécution

Running

wait

4.2.2. Définition de la sémantique opérationnelle.

Une solution pour modéliser le comportement d'une tâche a été envisagée dans [BEN06] dans le domaine de la robotique mais tout en utilisant un digramme UML à aspect statique. Elle consiste à illustrer la dépendance d'une tâche avec une autre (Precedes). En effet, une contrainte sur le démarrage ou la fin de la seconde activité est précisée : start-to-start, finish-to-start et finish-to-finish (PrecedenceKind). L'exécution d'un processus consiste à réaliser toutes les tâches qui le composent. Le processus est donc terminé quand toutes les tâches qui le composent sont achevées. Une tâche ne peut être commencée que si les tâches dont elle dépend sont commencées (précédence start-to-start) ou terminées (précédence finish-to-start). Au fur et à mesure du développement, le taux de réalisation d'une activité augmente jusqu'à ce qu'elle soit terminée (voir figure 15). Elle ne peut être terminée que si les tâches précédentes de type finish-to-finish sont terminées. C'est le développeur qui décide de quelle tâche commencer, continuer ou terminer.

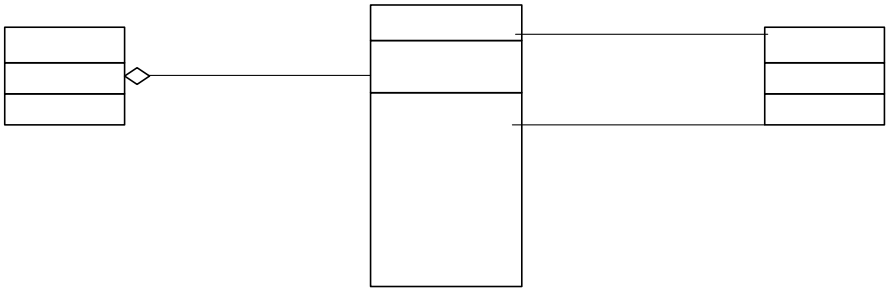

Figure 15*: *Modèle étendu pour la définition de la sémantique opérationnelle

Pour compléter la sémantique opérationnelle, il est nécessaire de compléter le modèle précédent avec les informations liées à l'état des tâches. Un attribut *progress* est ajouté sur la méta classe task. Il présente le taux de progression d'une tâche : non commencée (-1), en cours (0..99), ou terminée (100).

5. Modélisation de l'Ordonnanceur

L'ordonnanceur est considéré comme étant une pièce fondamentale d'un système temps réel. Il est en charge de définir le séquencement possible d'un ensemble de tâches exécutables par un

processeur. Par « possible » on veut dire que l'ordonnancement prend en compte les contraintes de dépendances (temporelles ou causales) et d'échéance aux quelles les tâches en question sont soumises.

Pour modéliser cette composante indispensable de l'RTOS, plusieurs diagrammes peuvent être utilisé tels que le diagramme de classe servant à la description de l'ordonnanceur, le digramme de séquence modélisant un algorithme d'ordonnancement.

5.1. Définition de l'Ordonnanceur

Suite à la séparation de la structure de l'RTOS et de l'Ordonnanceur lors de la modélisation dans [SHO04], le modèle relatif à l'Ordonnanceur mentionnée dans la figure 16 est présenté par les entités suivantes :

- **schedulingPolicy** *:* Cette entité définit la politique d'ordonnancement utilisé par l'RTOS pour élire le processus à exécuter. Cette politique est basée sur la priorité et le deadline des tâches.
- **schedulableEntity** *:* Chaque entité présente une réponse spécifiant la concurrence d'une tâche en cours d'exécution et un trigger indiquant combien de fois chaque tâche devrait être exécutée.
- **executionEngine** *:* Il s'agit d'une ressource active et protégée de type processeur réalisant l'ordonnancement. Elle spécifie sa politique d'ordonnancement comme par exemple le FIFO ainsi que le contexte selon le temps (c'est-à-dire le temps nécessaire pour copier le contexte de la tâche suspendue depuis les registres processeurs vers la mémoire et lancer la tâche à exécuter). Elle représente la capacité du système pour réaliser une tâche.
- **TRessource** *:* Elle est accédée durant l'exécution d'une entité de type schedulableEntity. Elle spécifie la politique de contrôle pour répondre à la demande de l'entité schedulableEntity.
- **Trigger** *:* Il spécifie l'occurrence de l'événement qui cause l'exécution d'une entité de type schédulableEntity
- **realTimeSituationContext** *:* Cette entité fournit le contexte d'analyse

- **Contexte** *:* C'est le comportement qui peut être caractérisé par ses propres exigences en terme de qualité de services. Il possède les attributs suivants : Entry, activate et terminate.

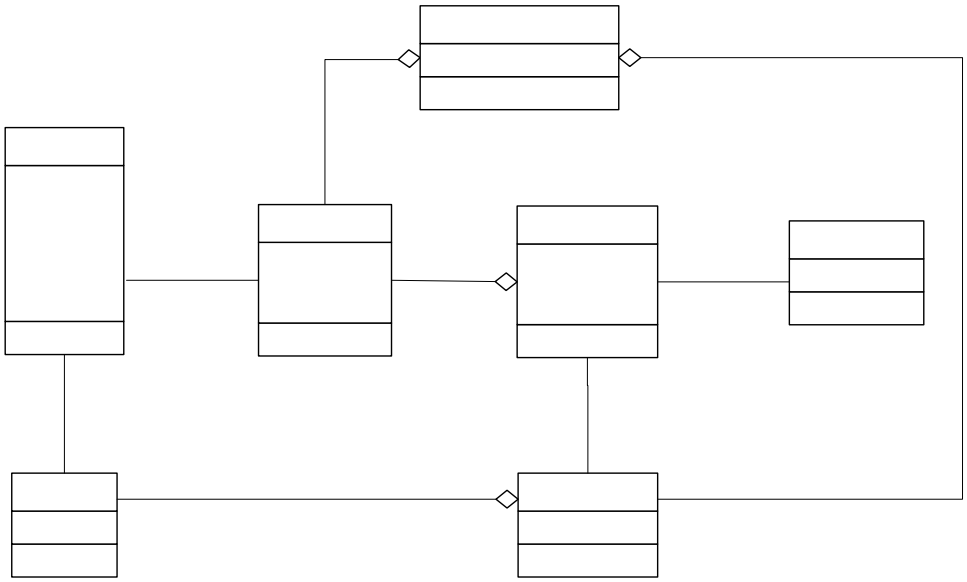

Figure 16**:** *Diagramme de classe correspondant à la définition de l'Ordonnanceur*

Le diagramme utilisé précédemment est le diagramme de classe. Ce diagramme ne peut fournir qu'une vue statique du système, il est en fait incapable de donner une vue comportementale du système. Par ailleurs, pour combler ce manque, une description à l'aide des machines à états finis (Final state machine FSM) a été associé à l'entité Task dans [SHO04].

5.2. Modélisation d'un algorithme d'ordonnancement

Dans [MAR06], une proposition consistait à modéliser un algorithme d'ordonnancement en utilisant le diagramme de séquence. La figure 17 présente la modélisation de l'algorithme d'Ordonnancement Rate Monotonic. En effet, Rate Monotonic scheduling [CLL73] est un algorithme d'ordonnancement temps réel en ligne à priorité constante. Il attribue la priorité la plus forte à la tâche qui possède la plus petite période. C'est un optimal dans le cadre d'un système

tâches synchrones, indépendantes et à échéance sur requête avec un Ordonnanceur préemptif. De ce fait, il n'est généralement utilisé que pour ordonnancer des tâches vérifiant ces propriétés.

Pour mettre l'accent sur l'aspect temps réel lors de l'ordonnancement, une intégration d'un certain nombre de stéréotypes a été faite tels que :

- Le stéréotype "SaResource" du profil SPT représente un genre de ressource protégée (par exemple, un sémaphore) qui est consultée pendant l'exécution d'une tâche. Il peut être partagé par des actions concurrentes multiples, comme il doit être protégé par un dispositif de verrouillage. L'étiquette "SAaccessControl" représente la politique de contrôle d'accès pour manipuler des demandes des travaux d'établissement du programme (dans notre modèle, ` PriorityInheritance ').
- Le stéréotype "SAschedRes" du profil SPT représente une tâche. La Tâche a été indiquée en tant qu'active où chacun de ses instances (T1, T2 et T3) a son propre « thread » et peut lancer une activité.
- Le stéréotype "SAscheduling" du profil SPT représente un Ordonnanceur qui est responsable de traiter les demandes d'un service de la part des clients. Ce stéréotype est basé sur la politique appropriée de contrôle d'accès pour ce service. Si un e service est occupé, alors la réponse peut demeurer en suspens jusqu'à ce que l'accès soit possible. L'étiquette "SAschedulingPolicy" représente alors l'ensemble de règles pour assigner le temps processeur à un ensemble des tâches à exécuter par le système.

Pour mettre l'accent sur la relation entre tâche/ressource, un certains nombre d'attributs et de méthodes est rajouté au modèle d'ordonnancement des tâches tels que :

- Idle : pour dire qu'un processus est en repos
- Busy : lorsqu'il s'agit d'une ressource occupée
- Delayed : lorsqu'une tâche dépasse le delai
- Release : libérer une ressource
- Awake : tâche réveillée
- Assign : attribuer

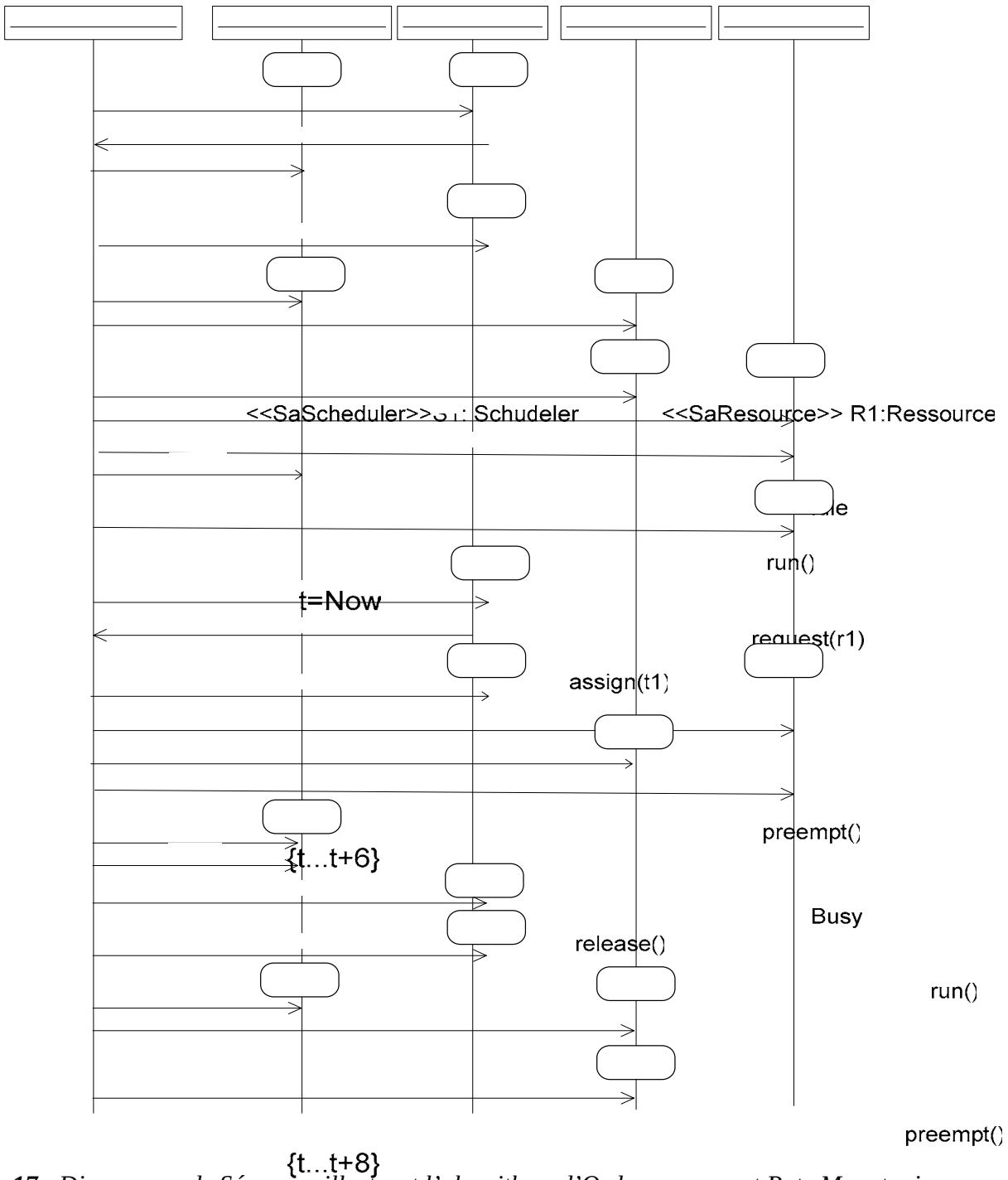

Figure 17 : *Diagramme de Séquence illustrant l'algorithme d'Ordonnancement Rate Monotonic*

L'handicape de cette proposition réside au niveau de l'existence d'un grand nombre d'algorithmes d'ordonnancement, et par suite le designer va se trouver en premier lieu face à la modélisation de plusieurs algorithmes d'Ordonnancement à l'aide d'une suite de diagrammes de Séquence et en deuxième lieu, face à la problématique d'intégration de l'ensemble de ces diagramme dans le processus MDA.

6. Modèles proposés

Nous rappelons que l'approche descendante utilisée dans [SHO04] fait une séparation explicite lors de la modélisation de la structure de l'RTOS et de l'ordonnanceur sans définir les sémantiques temporelles. Cette approche se base sur deux diagrammes à aspect statique.

Pour l'approche ascendante issue de [DAV06], elle se base sur le même diagramme et ne touche pas l'aspect comportemental.

Pour notre approche, nous décrivons la structure de l'RTOS à travers un diagramme de classe qui inclut la définition de la sémantique opérationnelle pour présenter la structure de l'RTOS. Nous définissons ensuite le comportement d'une tâche, afin d'assurer la cohérence entre les différents diagrammes UML. Pour aboutir au modèle d'ordonnancement, nous définissons les variations sémantiques temporelles et transitionnelles relatives aux différents états d'un processus temps réel.

Le tableau 4 donne une étude comparative entre différentes approches qui peuvent être exploitées lors de la modélisation d'un RTOS déjà citées précédemment, il permet de positionner notre démarche par rapport à elles.

Modèle	Description de la structure de l'RTOS	Description de l'Ordonnanceur
OSEK	▪ Utilisation du diagramme de classe ▪ Modélisation de la variation des états d'une tâche à l'aide des FSM ▪ Manque de cohérence entre les diagrammes	▪ Diagramme de classe ⟶ Aspect statique ▪ Indépendance entre la description de l'Ordonnanceur et la structure de l'RTOS
VxWorks	▪ Utilisation du diagramme de classe ▪ Entités séparées ▪ Possibilité d'intégration d'attribut couvrant l'évolution dans le temps ▪ Possibilité d'utilisation des relations de réciprocité	▪ Absence totale de l'aspect comportemental ▪ Pas de définition de modèle d'ordonnancement
Approche proposée	▪ Utilisation du diagramme de classe ▪ Définition des états possibles d'une tâche à l'aide des statecharts ⟶ Garantir la cohérence entre les diagrammes	▪ Définition et implantation des variations sémantiques associées aux statecharts

Tableau 4 : *Bilan récapitulatif des différentes approches de modélisation des RTOS*

6.2. Modèle associé la structure d'un exécutif temps réel

Deux diagrammes sont proposés pour la description de la structure de l'ordonnanceur, un diagramme de classe décrivant les principales composantes de l'RTOS, et un diagramme d'états transition est associé à l'entité tâche pour modéliser son aspect comportemental.

Nous nous basons alors sur le modèle présenté dans la figure 11 tout en ajoutant quelques attributs tels que periode, dateFirstActvation, deadline et duration au niveau de l'entité Task. Nous tenons à respecter notamment la spécification d'un exécutif temps réel présentée au niveau du premier chapitre. Nous intégrons aussi dans notre modèle d'autres entités telles que Process, Prcedes qui sont déjà mentionnées au niveau de la figure 15. Le modèle final issu est alors conforme à la figure 18.

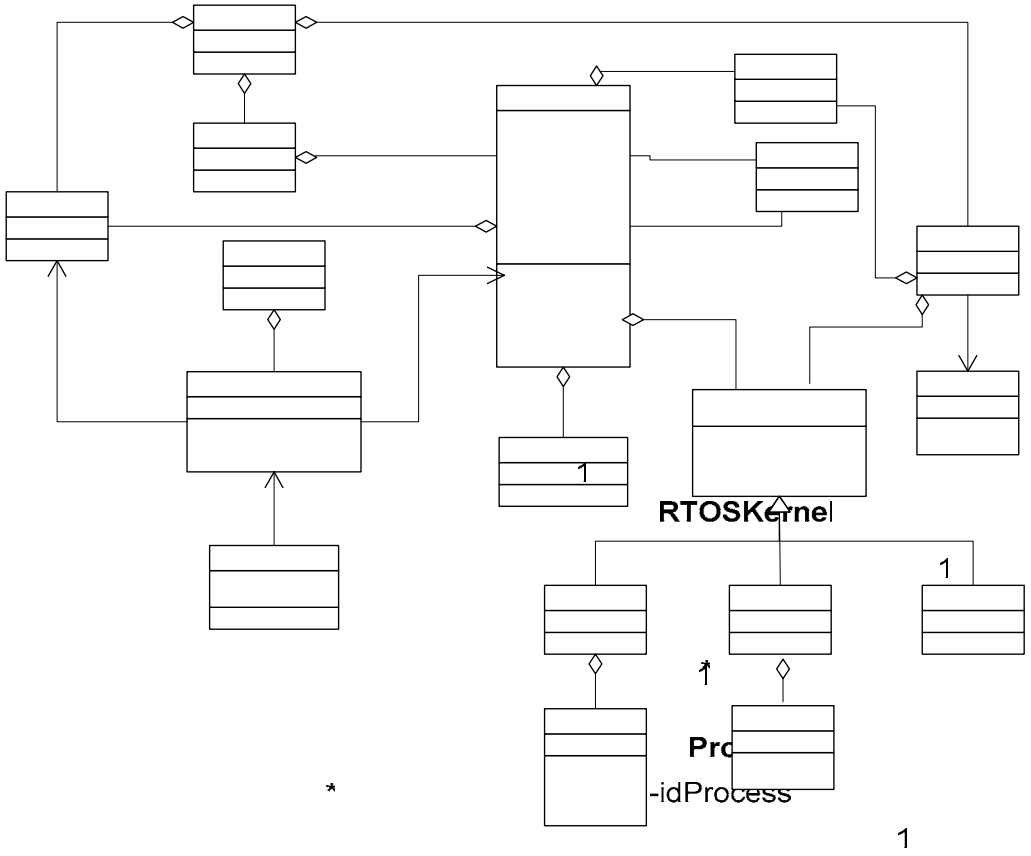

***Figure 18** : Modèle statique de la définition de la structure de l'RTOS proposé*

Vu qu'un patron de conception ou encore un design pattern est la formalisation d'une approche pour résoudre un problème commun dans un contexte particulier [Dou98] et que son utilisation lors de la conception d'un logiciel permet de réutiliser des solutions construites et validées pour des problèmes similaires, trois patrons pour la fiabilité et la sécurité des systèmes temps réel peuvent être utilisés :

- Patron chien de garde : Les chiens de garde permettent d'associer une routine à l'interruption de l'horloge du système. Ainsi, la routine s'exécute une fois le délai expiré.
- Patron de surveillance et action : Ce patron possède un système d'action (qui effectue les actions destinées à contrôler un processus), un système de surveillance (qui garde une

trace de ce que le système d'action doit faire et qui surveille l'environnement physique de l'application) et un système qui contrôle les deux premiers systèmes

- Patron de directives de sécurité : Le patron de directives de sécurité utilise un coordinateur central pour la surveillance de la sécurité et la tolérance aux fautes du système. Il fonctionne comme un chien de garde intelligent qui traque et coordonne toute la surveillance du système en saisissant les informations relatives aux délais des chiens de garde, aux erreurs logicielles et aux fautes détectées par les systèmes de détection du patron de surveillance et action. En cas de dysfonctionnement, le système des directives de sécurité agit de façon à remplacer le sous-système défaillant par des actions de recouvrement effectuées par un système redondant

Nous associons à l'entité Task un diagramme d'états transitions décrivant une vue comportementale de l'état d'une tâche, au cours de la deuxième phase de notre proposition. Nous exprimons quelques règles OCL relatives à chaque état dans le but d'atteindre la qualité correction du comportement du système (voir figure 19).

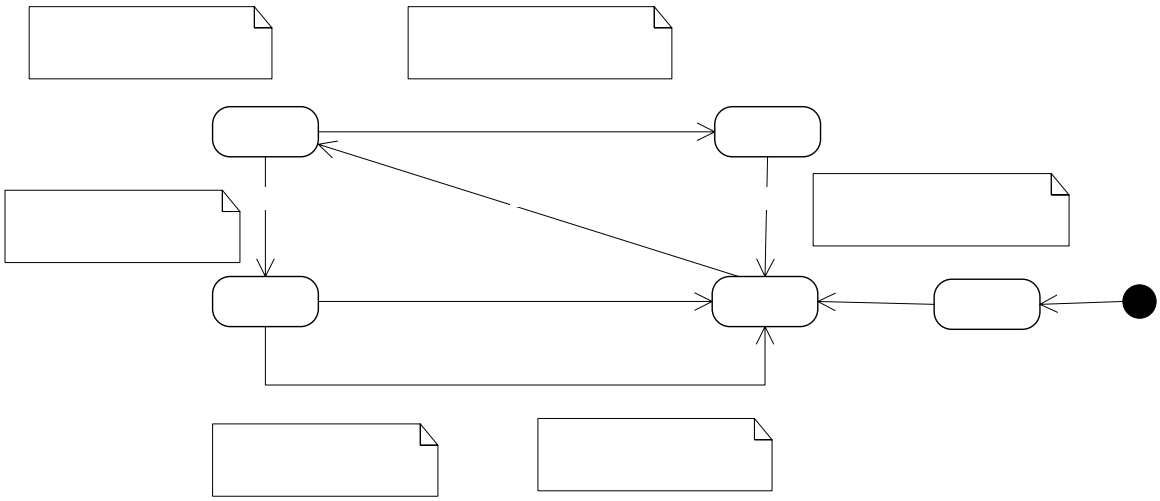

Figure 19**:** *Diagramme d'états transitions relatif à l'entité tâche annoté avec des contraintes OCL*

6.3. Modèle associé à l'ordonnanceur

Nous rappelons que nous avons déjà utilisé un diagramme de classe pour la définition des différentes composantes de l'RTOS. Nous avons associé à l'entité Task, qui constitue le cœur du modèle, un diagramme d'états transitions. Vu que ce diagramme présente des points de variations sémantiques dont leurs définitions sont laissées à la charge du designer, nous proposons de les préciser et de les expliciter ultérieurement dans la phase de définition des sémantiques temporelles. Cette phase qui nous a amenée au modèle d'ordonnancement, sera détaillée dans le chapitre suivant.

7. Conclusion

La conception d'application selon l'approche MDA débute par l'analyse du problème, modélisé d'un point de vue abstrait. MDA définit pour cette phase la notion de PIM. Cette étape de modélisation sur laquelle nous avons mis l'accent, s'appuie sur le langage UML. Nous avons plus particulièrement focalisé notre étude sur deux des diagrammes les plus représentatifs : le diagramme de classes et celui d'états transitions. Le digramme de classe décrit la structure d'un RTOS. Un diagramme d'états transitions est associé à l'entité Task décrivant l'évolution de son état au cours du temps. Nous lui définissons dans le chapitre suivant la variation des sémantiques dans le but de caractériser l'Ordonnanceur.

C H A P I T R E

5

Implantation des statecharts et Génération de Code

1. Introduction

Dans le chapitre précédent, nous avons étudié la façon dont nous spécifions en UML la structure d'un RTOS associé à un modèle d'application. A partir du digramme de classe, nous définissons les états possibles de l'entité « Task » sous forme d'un diagramme d'états transitions. Dans ce chapitre, nous définissons les points de variations sémantiques que présente ce diagramme tout en ajoutant des extensions sous la forme d'un profil qui contient des stéréotypes utilisés pour l'annotation des éléments du modèle. Ces annotations, qui décrivent des informations liées à la définition de la sémantique temporelle et transitionnelles.

Une fois cette tâche accomplie, des solutions techniques à savoir l'énumération et la réification doivent être mises en place pour permettre l'implantation des statecharts. Le modèle final issu va correspondre au modèle cible lors de la transformation des modèles.

Toujours dans une même optique de la démarche MDA, nous finissons par la génération automatique du code.

2. Modélisation des variations sémantiques

Dicté par son caractère généraliste, UML définit un ensemble de points sémantiques non définit [ARN07] explicitement. C'est alors dans le contexte d'une méthodologie donnée, et s'appuyant sur UML comme langage de modélisation, que le sens de chacun des points de variation sémantique doit être précisé.

Les points ouverts de variations sémantiques doivent être précisés clairement et volontairement par tous les outils et méthodes affirmant s'appuyer sur UML car dans le cas contraire, il ne serait pas possible de conclure sur le sens réel d'un quelconque modèle UML

Le tableau suivant issu de [FRA04] synthétise les différentes variations sémantiques des statecharts tels qu'ils sont décrits dans UML 2.0.

Variations	Type	Dépendances	Portée
TimeOut	syntaxique	Sémantique temporelle	Transition
Transitions inter-niveaux	syntaxique	---	Automate
Référence à un état	syntaxique	---	Transition
Evénement conditionnel	syntaxique	---	Transition
Etat historique	syntaxique	---	Automate
Sémantique temporelle	sémantique	---	Application
Sélection des événements	sémantique	---	Automate
Durée des événements	sémantique	Evénement différé	Automate
Priorités entre transition	sémantique	Plus interne	Automate
Non déterminisme	Sémantique	---	Automate

Tableau 5 : *Récapitulatif des différents points de variations [FRA04]*

A fin de spécifier les différents points de variations associées aux statecharts, il est nécessaire de disposer d'un méta-modèle permettant d'exprimer les différents choix possibles. Ce méta-modèle est illsustré par la figure 20. La question de la sémantique temporelle est représentée par l'élément *TimeProgression* qui peut prendre deux valeurs : synchrone ou asynchrone. Les variations concernant la gestion des évènements sont représentées par les éléments *EventSelection* et *EventChoice* montrant respectivement la sélection de l'événement et son choix.

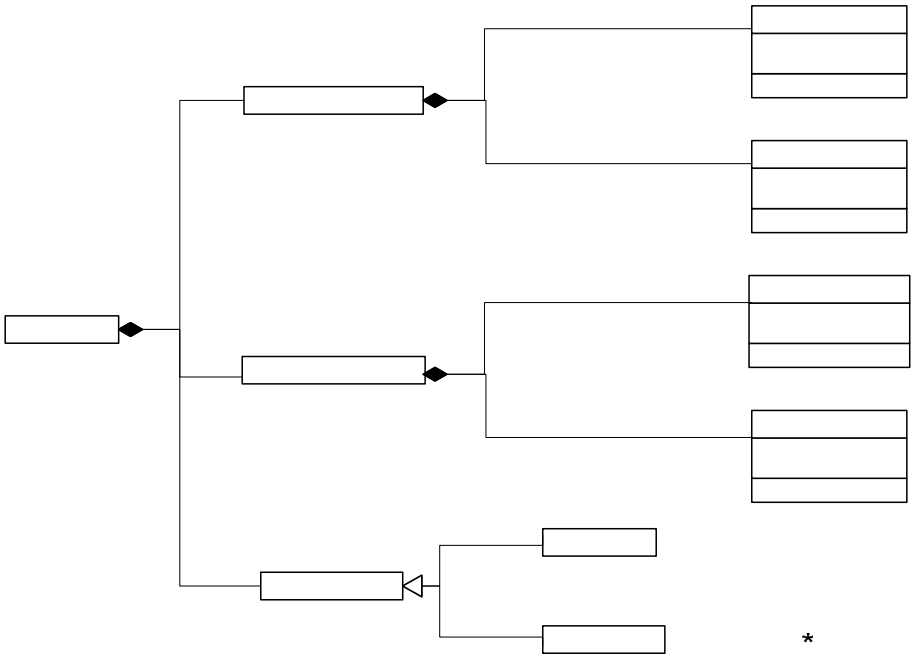

Figure 20 : *Récapitulatif des différents points de variations [FRA04]*

En ce qui concerne les variations relatives aux transitions, elles sont présentées par les éléments *TransitionSelection* et *TransitionChoice*. L'avantage de cette solution est qu'elle permet une grande flexibilité dans la définition de la sémantique associée aux statecharts. En effet les portions de code associées à la progression de l'automate ne sont pas figées et peuvent être facilement modifiées dans le modèle adéquat. La sémantique des statecharts est résumée en une procédure, appelée *step()* qui correspond à la réaction de l'automate face à l'occurrence d'un évènement. Cette procédure peut se résumer à trois actions principales : sélectionner un évènement, sélectionner une transition parmi celles que déclenche l'évènement et tirer la transition choisie.

```
procedure step()
begin
eventSet := eventPool.select();
anEvent := eventSet.choice();
transitionSet := getFirableTransition(event).select();
aTransition := transitionSet.choice();
```

```
aTransition.fire();
end.
```

Figure 21*: *La procédure Step en pseudo-code[FRA04]

D'après la figure 21, la procédure *step* exige la sélection d'un élément dans un ensemble en fonction de différents critères de priorités. Dans la cas des évènements par exemple, les différents critères de priorités (évènements internes/externes, ordre dans la file, etc.) permettent de sélectionner un sous-ensemble d'évènements. Il peut être nécessaire cependant d'avoir à départager plusieurs évènements parmi ce sous-ensemble. Deux politiques peuvent être évoquées: soit le choix de l'évènement se fait de façon aléatoire, par un tirage au sort par exemple, soit de façon arbitraire si l'on fixe l'évènement choisi.

2.1. OCL pour qualifier les points de variation sémantique

Les différentes étapes de la procédure *step()* définissent le noyau de la sémantique des statecharts utilisé dans UML. Chacune de ces étapes correspond à une méthode qui peut être caractérisée à l'aide du langage OCL.

La sélection des évènements peut-être envisagée de nombreuses façons : FIFO, LIFO . . . . Pour notre cas il s'agit d'une file de priorité : l'événement tiré est celui ayant la priorité la plus haute. Quelle que soit la politique choisie, la procédure manipule toujours les mêmes éléments : un ensemble d'évènements sur lequel elle en extrait un sous ensemble (voir figure 22).

```
context EventManagement
def : eventPool : Set
inv : eventPool->select(evt : Event | event->isOver())->isEmpty()
context : PRIORITY::nextEvent() : Set
pre : eventPool->notEmpty()
post : return = self.getAllAttributes()-> select(name='priorite').upper
context : LIFO::nextEvent() : Set
pre : eventPool->notEmpty()
post : return = eventPool@pre->asSequence()->last()
context : FIFO::nextEvent() : Set
pre : eventPool->notEmpty()
post : return = eventPool@pre->asSequence()->first()
```

***Figure 22 :** Contraintes OCL sur la gestion des évènements*

2.2. Un Profil UML pour spécifier les choix sémantiques

Dans le cadre de notre application, un profil UML va permettre de définir une extension d'UML autorisant l'utilisateur à décorer les statecharts avec les propriétés adéquates. Le profil UML nécessaire doit permettre de préciser les éléments qui déterminent le comportement de la procédure *step()*. Il faut donc spécifier comment sélectionner un évènement, comment sélectionner une transition, et quelle est la sémantique associée à la progression de l'automate. La figure 23 ci-contre présente une représentation de ce profil.

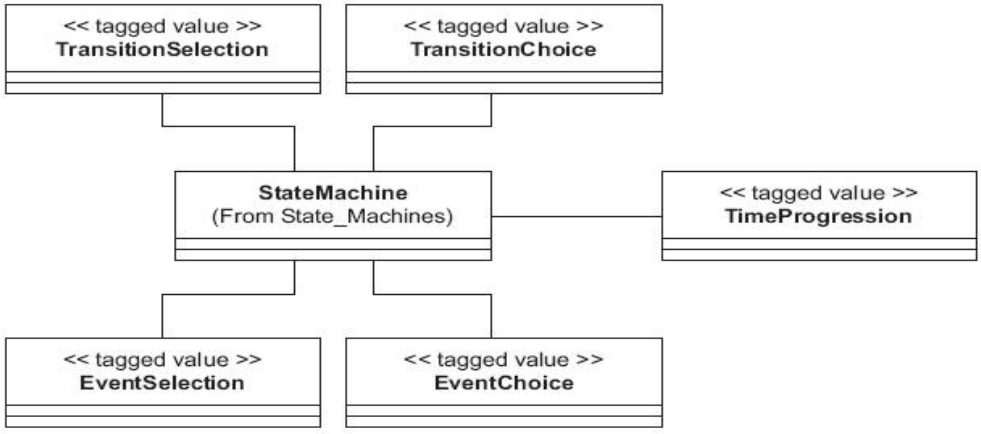

Figure 23 : *Profil UML pour préciser les choix sémantiques associés aux statecharts [FRA04]*

Cependant les tagged values ne suffisent pas. En effet, une telle étiquette n'est qu'une paire (nom, valeur) qui ajoute une nouvelle propriété à un élément de modélisation. Ce système ne permet pas de faire le lien entre la sélection des évènements dans la pile et le code associé à cette opération. Pour cela, il est donc nécessaire de décrire dans un modèle spécifique le code de ces opérations.

Les modèles correspondant au méta-modéle proposent donc un certain nombre de solutions pour chacune des étapes de la procédure *step()*. Chaque solution est identifiée par un nom, et c'est ce nom qui sera associé à la tagged value correspondant dans le modèle.

3. Implantations des statecharts

Comme nous venons de dire la spécification des choix sémantiques est insuffisante. De ce fait, l'implantation des statecharts constitue une phase primordiale au cours de notre démarche

proposée. Le modèle final issu, après cette étape, correspondra au modèle cible lors de la transformation des modèles.

3.1. Techniques d'implantation des statecharts

Un ensemble d'approches [LUI03, PIN04] a été proposé dans la littérature afin d'implémenter les statecharts, à savoir la spécialisation de fonctions et l'utilisation d'interpréteur générique muni d'un ensemble de structures de données spécifiant le comportement. Ces approches sont gourmandes en terme d'utilisation du processeur et de la mémoire. Elles sont aussi difficiles à maintenir.

Pour notre application nous optons pour la technique la plus simple, celle basée sur l'énumération et la réification. Plusieurs designs pattern sont spécialisés pour la mise en œuvre de ces techniques.

En effet, la réification consiste à matérialiser un concept par un objet et de le manipuler concrètement. Il s'agit de transformer une entité qui n'est pas un objet en objet. Elle peut être utile dans les méta-applications. Cette technique présente un outil graphique et mathématique de modélisation. Dans le cas des STRE, ils restent restreints pour la modélisation de l'aspect concurrence et ordonnancement de l'RTOS. L'énumération consiste à attribuer les valeurs énumérées que peut prendre un concept

3.1.1. Enumération des états et des événements

L'état de l'entité Task prend ses valeurs de l'ensemble :{created, waiting, ready, running, stopped}. Pour la représentation des événements, nous faisons recours à une méthode processEvent (evt :Event) où evt est un type énuméré. Pour la représentation de la réaction à un événement, l'objet Task est doté d'une méthode processEventPlay qui détermine le comportement en fonction de l'état courant d'une tâche.

3.1.2. Réification des événements

Le concept de réification peut être appliqué aux différents évènements au niveau des statecharts. En effet, chaque évènement est considéré comme action de l'automate. Vu que le pattern Command [FOW97] permet de spécifier, stocker et exécuter des actions à des moments différents (les commandes exécutées peuvent être stockées ainsi que les états des objets affectés), il peut

être utiliser pour décrire les événements agissant sur l'état d'un processus. Le résultat de la réification des éventements est illustré par la figure 24.

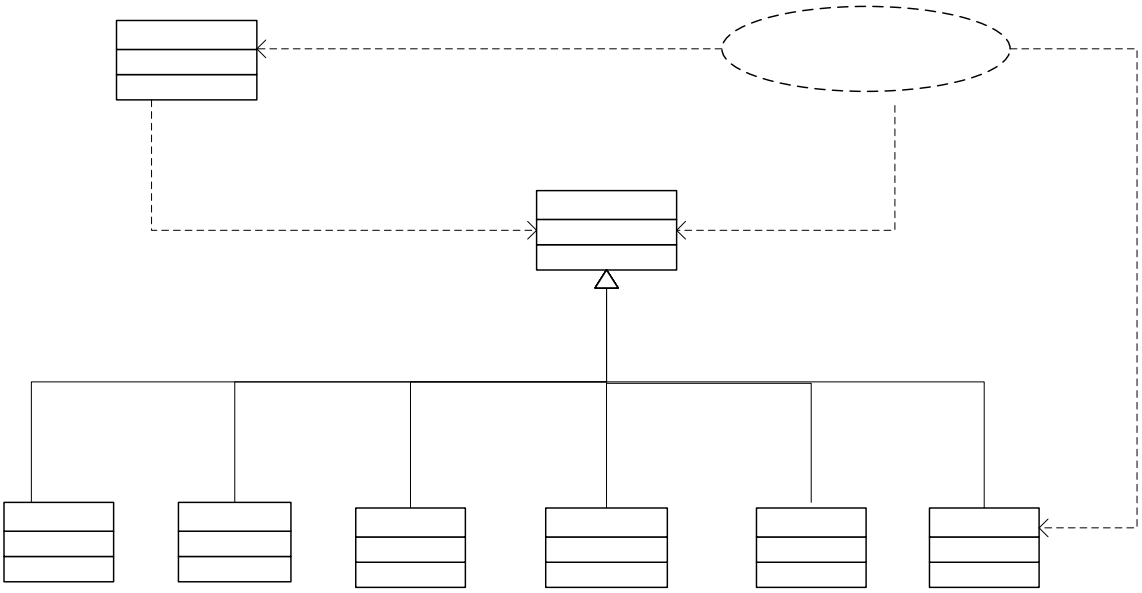

Figure 24 : *Application du pattern command sur l'entité Task*

3.1.3. Réification des états

Le principe de la réification des états consiste à séparer le comportement lié à un état dans objet. Le pattern State [FOW97] semble une solution efficace pour notre cas (voir figure 25). En effet ce pattern permet, lorsqu'un objet est altéré de changer son comportement changé. Le changement d'état peut parfois poser des problèmes dans leurs gestions, le Pattern State permet de pallier à ce problème de manière simple et rapide.

State doit comporter des classes précises participantes à ce Pattern :

- Context
- State
- ConcreteState

On utilise State lorsque :

- Le comportement d'un objet dépend de son état, qui change à l'exécution.
- Les opérations sont constituées de parties conditionnelles de grande taille.

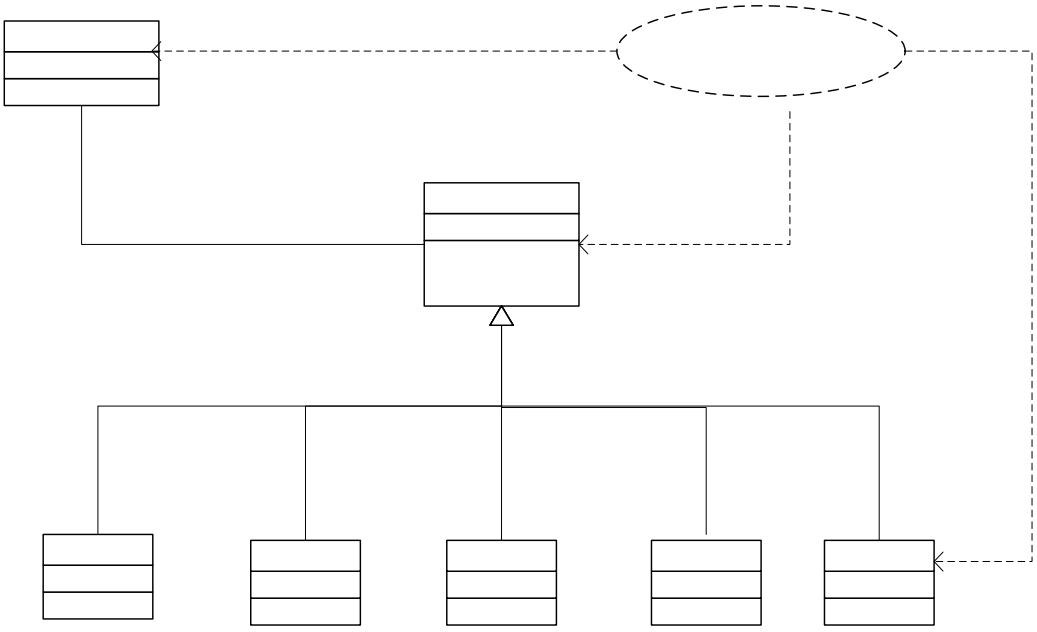

Figure 25 : *Application du pattern State sur l'entité Task*

3.1.4. Réification des états et des évènements

Il s'agit lors de cette phase de réifier les états et les événements en même temps et ce en appliquant les deux patterns State et Command. Cette solution représente d'une façon souple de l'automate mais augmente énormément le nombre de classes. La réification des états et des événements nous amène au modèle présenté par à la figure 26.

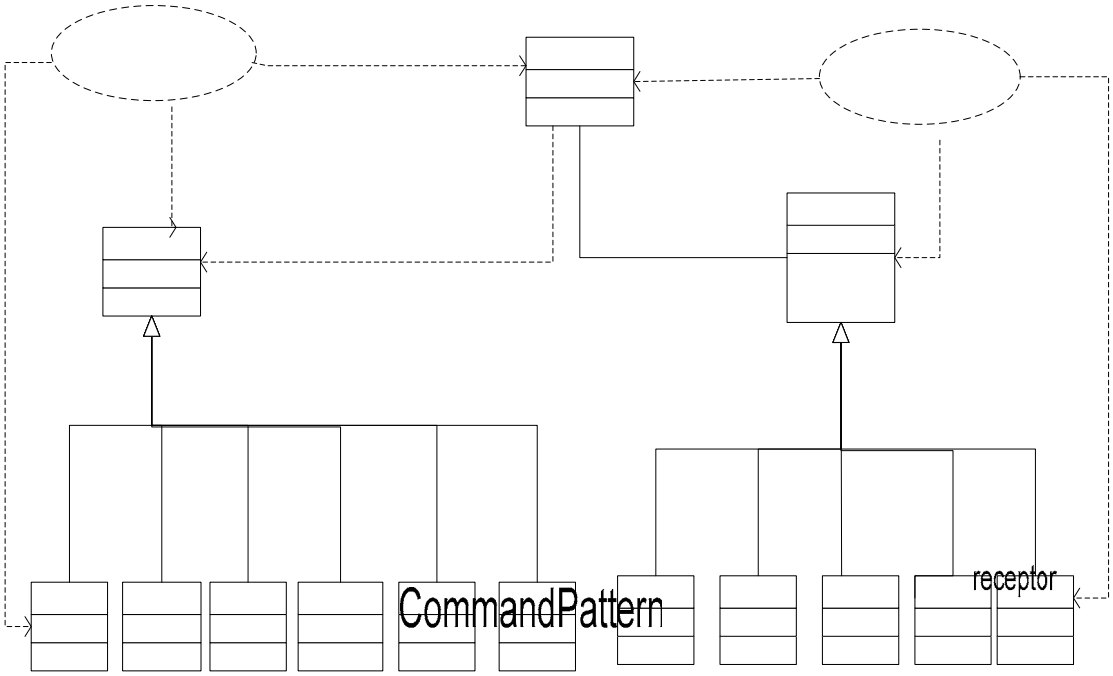

Figure 26 : *Application du pattern state et du pattern Command sur l'entité Task*

3.2. Progression de l'automate

A la lumière des solutions présentées précédemment et en vue d'assurer la progression de l'automate, il est nécessaire de mettre l'accent sur l'aspect déterministe du système, c'est-à-dire il est indispensable de déterminer l'état courant de l'automate et le comportement à adopter en fonction de l'événement survenu.

Quand il s'agit de l'énumération des états et des événements, le code réagissant la progression de l'automate est localisé dans la méthode processEvent(). Quant à l'énumération des états et la réification des événements, le code sera reparti entre la méthode processEvent() et execute() de chaque classe. S'il s'agit de la réification des états et l'énumération des événements, le code sera réparti entre la méthode processEvent() et la méthode processEventPlay() de chaque classe état. Finalement, lorsque nous réifions les états et les évènements, Le code est réparti entre la méthode

processEvent() de la classe principale, les méthodes processEvent() des classes états et les méthodes execute() des classes évènements.

Les solutions de réifications et d'énumérations ne nous permettent pas également de représenter la notion de file de messages relative à la progression de l'automate. Le temps n'est pas pris en considération. Pour surmonter ce problème, l'utilisation du patron active Object [FOW97] est alors indispensable.

En effet ce patron dissocie la tâche qui reçoit une requête de celle qui la traite. La mise en oeuvre des différentes stratégies d'affectation peut ainsi être réalisée indépendamment du fonctionnement des clients des objets actifs (qui se contentent de remettre un travail à effectuer à un relais offrant la même interface que l'objet réel) et de celui des objets réels qui exécutent ces travaux dans une tâche spécifique. Ce patron est donc efficace pour la réalisation des différentes politiques de parallélisme.

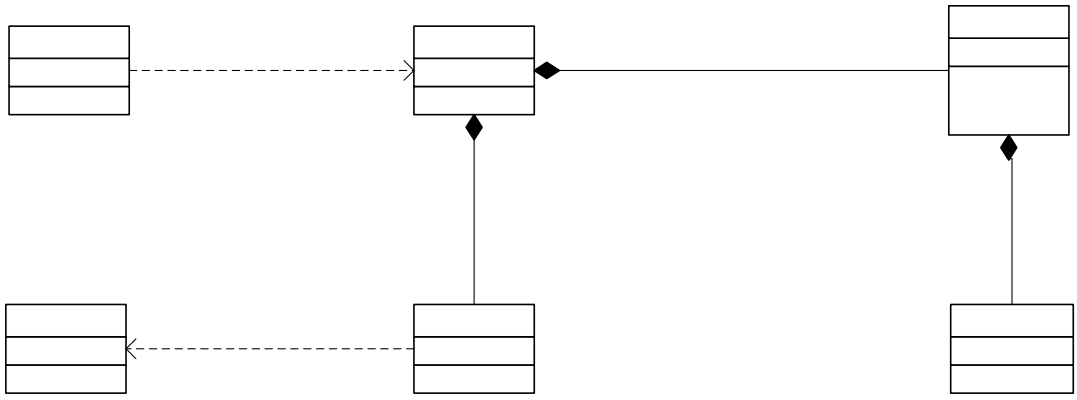

Figure 27 : *Application du pattern Active Object sur l'entité Task*

3.3. Modèle final

Suite à l'application de la réification des états et des événements, ainsi que l'illustration de l'évolution de l'automate, le modèle final correspondant au modèle cible au niveau PIM de l'approche MDA est représenté par le modèle ci-dessous :

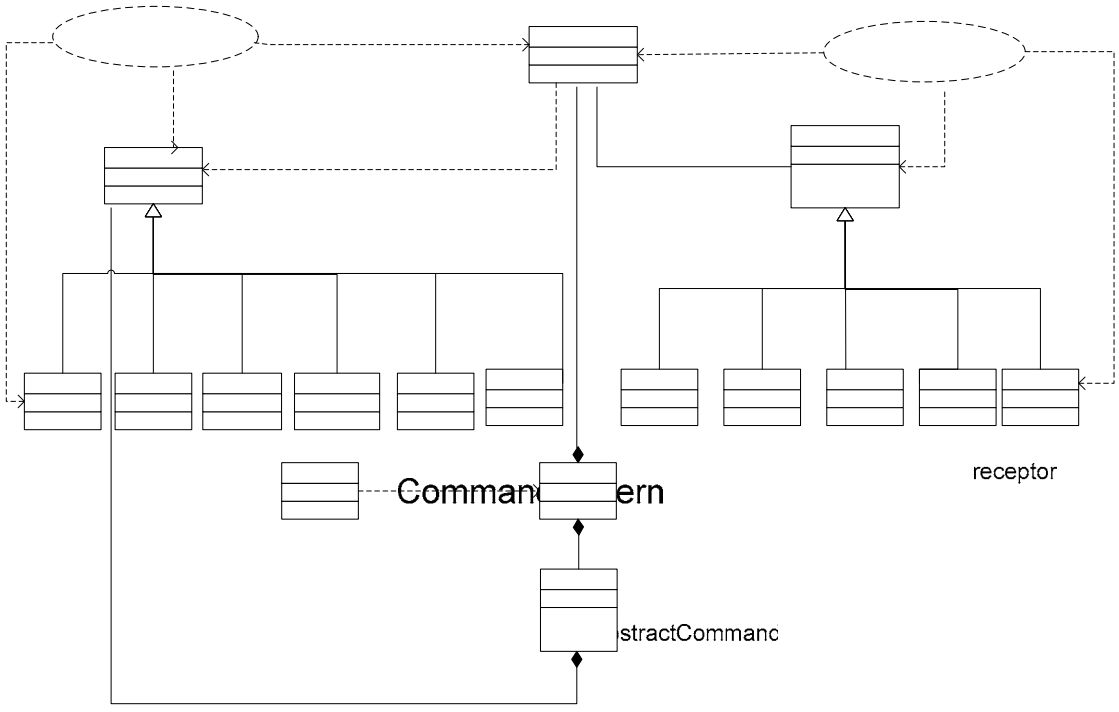

Figure 28 : *Modèle d'ordonnancement issu de l'implémentation des statecharts*

Ce modèle d'ordonnancement, ainsi que le modèle proposé pour la description de l'RTOS mettent en œuvre l'adéquation Algorithme Architecture. En effet, si ces deux modèles sont intégrés à un profil déjà existant et tenant compte de l'application et de l'architecture, cette adéquation sera explicite. La communication entre l'application et l'RTOS et l'application sera assurée via l'entité Task. Pour la communication entre l'RTOS et l'architecture, elle sera identifié grâce aux entités ressource et horloge.

4. Génération de Code

Notre objectif consiste à transformer un modèle source XML obtenu automatiquement à partir de notre modèle source, en un modèle cible XML. Pour réaliser les transformations, nous nous appuyons sur un modèle de transformation en langage ATL. Pour décrire le modèle à transformer, nous utilisons le langage KM3, qui permet de définir des modèles selon le méta-modèle MOF sous une forme textuelle simplifié.

Il est à noter que l'exemple utilisé lors de la transformation est pris adéquatement étant donné que l'objectif de notre travail est de montrer juste la faisabilité de l'utilisation de l'IDM pour l'intégration de la modélisation de l'RTOS lors de la conception des STRE.

4.1 Modèle source

Le modèle source que nous transformons correspond au diagramme de classe présenté par la figure 18. Le code correspondant en XMI, basé sur XML offre une structure arborescente à notre modèle en présentant les classes et les attributs sous forme textuelle. Nous obtenons le code présenté ci-dessous :

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<xmi:XMI xmi:version="2.0" xmlns:xmi="http://www.omg.org/XMI" xmlns="Xml">

  <RtosKernel name="RTOSCES">
    <Process idProcess="P1" >
               <Task idTask="T1" priority="1" taskState="create"
dateFirstActivation="0" deadline="12" duration ="10" periode="2"
ResponseTime="12" Periodicity="1">
                  <TaskContext/>
                  <Ressource ressourceProprety="Processor" action="use"/>
                  <Prcedes kind="starttostart"/>
                  <MeanOfCommuinication>
                        <ProtectedVar>
                              <Mutex/>
                        </ProtectedVar>
                        <LettreBox>
                              <FileMessage/>
                        </LettreBox>
                        <TCPSocket/>
                  </MeanOfCommuinication>
                  <event nature="preemp"/>
            </Task>
    </Process>

    <ISR category="aa">
            <WatchDog/>
            <Ressource ressourceProprety="Processor" action="use"/>
                  <MeanOfCommuinication>
                        <ProtectedVar>
                              <Mutex/>
                        </ProtectedVar>
                        <LettreBox>
                              <FileMessage/>
                        </LettreBox>
                        <TCPSocket/>
                  </MeanOfCommuinication>
      </ISR>
      <event nature="preemp"/>
  </RtosKernel>
  <AlarmAction>
```

```
      <Alarm>
            <Counter maxAllowValue="30" minCycle="3"/>
             <Task idTask="T1" priority="1" taskState="create"
dateFirstActivation="0" deadline="12" duration ="10" responseTime="2"
periodicity="1" >
                  <TaskContext/>
                  <Ressource ressourceProprety="Processor" action="use"/>
                  <Prcedes kind="starttostart"/>
                  <MeanOfCommuinication>
                        <ProtectedVar>
                              <Mutex/>
                        </ProtectedVar>
                        <LettreBox>
                              <FileMessage/>
                        </LettreBox>
                        <TCPSocket/>
                  </MeanOfCommuinication>
                  <event nature="preemp"/>
            </Task>
            <event nature="preemp"/>
      </Alarm>
  </AlarmAction>
</xmi:XMI>
```

Figure 29 : *Modèle source en XMI*

Afin de modéliser notre méta-modèle source (métamétamodèlisation), ce dernier doit être écrit en KM3 (Kernel MetaMetaModel) déjà décrit dans le chapitre 3, et qui se base sur la notion de packages, classes et références qui seront par la suite manipulés par ATL. Ce méta-modèle est décrit à l'aide de la figure ci contre :

```
package RTOSStructure {

      class RtosKernel {
            attribute name : String;
            reference Process container : Process1 oppositeOf RTOS;

            reference ISR container : ISR1 oppositeOf RTOS11;
            reference event container : Event1 oppositeOf RTOS2;
      }

      class Process1 {
            attribute idProcess : String;
            reference RTOS[0-1] : RtosKernel oppositeOf Process;
            reference Task container : Task1 oppositeOf RTOSModel;

      }

      class Event1
      {
attribute nature : String;
            reference RTOS2[0-1] : RtosKernel oppositeOf event;
      }
```

```
	class ISR1 {
	attribute category : String;
		reference RTOS11[0-1] : RtosKernel oppositeOf ISR;
		reference WatchDog [1-*] ordered container: WatchDog1;
		reference Ressource [1-*] ordered container: Ressource1;
		reference MeanOfCommuinication [1-*] ordered container:
MeanOfCommuinication1;

	}
	class WatchDog1{}
	class Task1 {
		attribute idTask : String;
		attribute priority : String;
		attribute taskState : String;
		attribute dateFirstActivation : String;
		attribute deadline : String;
		attribute duration : String;
		attribute periode : String;
		attribute ResponseTime : String;
		attribute Periodicity : String;
		reference RTOSModel[0-1] : Process1 oppositeOf Task;
		reference TaskContext [1-*] ordered container: TaskContext1;
		reference Prcedes [1-*] ordered container: Prcedes1;
		reference Ressource [1-*] ordered container: Ressource1;
		reference MeanOfCommuinication [1-*] ordered container:
MeanOfCommuinication1;
		reference event [1-*] ordered container: Event1;


	}
	class TaskContext1
	{}
	class Prcedes1
	{
	attribute kind : String;
	}
	class Ressource1
	{
	attribute ressourceProprety : String;
	attribute action : String;
	}

	class MeanOfCommuinication1{
	    reference ProtectedVar [1-*] ordered container: ProtectedVar1;
	reference LettreBox [1-*] ordered container: LettreBox1;
	reference TCPSocket [1-*] ordered container: TCPSocket1;
	}
	class ProtectedVar1{
	reference Mutex [1-*] ordered container: Mutex1;
	}
	class Mutex1{}
	class LettreBox1{
	reference FileMessage [1-*] ordered container: FileMessage1;
	}
```

```
	class FileMessage1{}
	class TCPSocket1{}
	class AlarmAction {
	reference Alarm [1-*] ordered container: Alarm1;
}
	class Alarm1{
	reference Counter [1-*] ordered container: Counter1;
	reference Task [1-*] ordered container: Task1;
	reference event [1-*] ordered container: Event1;
	}
	class Counter1{
	attribute maxAllowValue : String;
	attribute minCycle : String;
	}

}

package PrimitiveTypes {
	datatype String;
}
```

***Figure 30** : Méta-modèle source en KM3*

4.2 Modèle cible

C'est le modèle que nous voulons obtenir après l'exécution des transformations appliquées sur le modèle source. Il doit être conforme au méta-modèle décrit dans la figure 31:

```
package RTOSSchudeler {

	class shudeler {
	reference proxy [1-*] ordered container: Proxy;
		reference task [1-*] ordered container: Task;
		reference eventpool [1-*] ordered container: EventPool;

	    }
	class Proxy {}

	class Task {
		reference abstractstate [1-*] ordered container: AbstractState;
		reference abstractevent [1-*] ordered container: AbstractEvent;
		}
	class AbstractState {
		 reference stopstate [1-*] ordered container: StopState;
		reference waitstate [1-*] ordered container: WaitState;
		reference readystate [1-*] ordered container: ReadyState;
		reference createstate [1-*] ordered container: CreateState;
		reference runnigstate [1-*] ordered container: RunnigState;
		}
	 class EventPool {
	  reference abstractevent [1-*] ordered container: AbstractEvent;
	  }
	  class AbstractEvent {
		reference createevent [1-*] ordered container: CreateEvent;
```

```
            reference waitevent [1-*] ordered container: waitEvent;
            reference PreeemtEvent [1-*] ordered container: PreeemtEvent;
            reference terminateevent [1-*] ordered container: TerminateEvent;
            reference activateevent [1-*] ordered container: ActivateEvent;
            reference startevent [1-*] ordered container: StartEvent;
            }
        class StopState{}
        class WaitState{}
        class ReadyState{}
              class CreateState{}
              class RunnigState{}
              class CreateEvent{}
              class waitEvent{}
        class PreeemtEvent{}
              class TerminateEvent{}
                          class ActivateEvent{}
                          class StartEvent {}
             }

package PrimitiveTypes {
      datatype String;

}
```

Figure 31 : *Méta-modèle cible en KM3*

Nous présentons maintenant un exemple de règle ATL écrite pour assurer la transformation, cette règle est utilisée pour assurer la génération automatique du modèle cible, elle est présentée par la figure 32.

```
rule RTOSModeling{
      from
            s : RTOSStructure!Task1
      to
            w : RTOSSchudeler!shudeler ()

}
```

Figure 32 : *Règle en ATL*

5. Conclusion

Dans ce chapitre, nous avons rappelés les points de variations sémantiques des statecharts. Nous avons présentés les différentes techniques nécessaires pour les implémenter. Nous avons adopté une démarche d'intégration de design pattern dans le but de réutiliser des composants logiciels existants et éprouvés, plutôt que de recréer de nouveaux modèles pour l'implémentation des statecharts.

Le modèle final relatif à la variation des différents états de l'entité «Task» correspond au modèle cible de la démarche MDA.

Nous avons en effet présenté les différentes étapes menant de l'élaboration du modèle cible de l’ingénierie dirigé par les modèles jusqu'à la génération du code final.

Conclusion

Le domaine des systèmes temps réel, et d'une manière générale celui du développement des RTOS, représente de vastes sujets d'étude, que nous avons souhaité réunir.

Plus particulièrement, nous avons exposé dans ce mémoire de mastère les travaux concernant notre approche pour la prise en compte d'exécutif temps réel lors de la conception d'un STRE en optant pour une approche orientée objet. Notre objectif a été d'étudier et de mettre en pratique un paradigme récent : l'Ingénierie Dirigée par les Modèles.

Nous avons présenté dans le Chapitre 1 le contexte scientifique dans lequel notre travail a été réalisé. Nous avons introduit les systèmes temps réel, en nous focalisant notamment sur l'étude des caractéristiques d'un RTOS.

Dans le deuxième chapitre, nous avons effectué un tour d'horizon sur les tendances passés et actuelles utilisées dans le domaine de conceptions des STRE. Au cours de ce chapitre, l'étude de l'existant nous a montré que de nombreuses solutions avaient été proposées pour répondre à chacun de ces problèmes. Nous avons détaillé plus précisément deux approches : la notion de profil UML, ainsi que la définition de la sémantique temporelle et transitionnelle.

De cette étude est ressorti que l'une des principales préoccupations dans ce domaine était l'intégration des caractérisations temps réelles comme le temps d'exécution et les contraintes temps réelles et par conséquent la prise en compte de l'RTOS relatif à l'architecture et l'application considérées.

En partant de ce constat, il nous a semblé judicieux d'apporter notre contribution en générant automatique un RTOS tout en utilisant une démarche orientée objet. Cette démarche est mise en évidence au niveau du troisième chapitre, nous avons notamment présenté les éléments de l'ingénierie dirigée par les modèles, et plus particulièrement les spécifications MDA mises au point par l'OMG.

Notre travail est découpé en deux parties, pour lesquelles nous avons également suivi une démarche proche du développement basé sur les modèles.

La première partie, décrite dans le chapitre 4, détaille notre solution. Nous avons tout d'abord décrit un prototype pour la modélisation de la structure de l'RTOS à l'aide d'un diagramme de classe qui correspond au modèle source pour la démarche MDA.

Au niveau du cinquième chapitre qui constitue la deuxième partie de notre travail, nous avons définit et implanté la variation des sémantiques associées aux statecharts relatifs à la l'état d'un processus dans le but de spécifier le modèle d'ordonnancement. Le modèle issu de cette phase correspond au modèle cible de la démarche MDA Nous avons notamment fini avec la génération automatique de code.

Enfin, du point de vue des perspectives, nous souhaitons adapter notre travail à d'autres domaines que ceux directement liés aux RTOS dédiés au domaine de l'automobile.

Nous pouvons enrichir le modèle de profil relatif au statecharts, grâce au mécanismes d'extensions propres à UML pour pouvoir concevoir tout un profil propre aux STRE intégrant ses différentes composantes : architecture (mono processeur, multi processeur voir même SoC), application et RTOS.

Ainsi, il serait intéressant d'appliquer notre approche au développement d'applications de domaines tels que celui de la robotique ou celui des réseaux de capteurs. Dans ces domaines, des entités de nature hétérogène sont amenées à communiquer, qu'il s'agisse de robots coopérants, et dont l'ensemble peut être vu comme un système réparti; ou qu'il s'agisse des réseaux de capteurs communiquant au sein d'un habitat « intelligent ».

La mise au point d'extensions à ce profil, contenant des stéréotypes propres à chacun de ces domaines, tirerait alors les mêmes avantages que ceux que nous avons montrés dans nos résultats.

Références

[ALA92] M. Alabau and T. Dechaize, "Ordonnancement temps réel par échéance". In T.S.I., volume 11. n.3, 1992.

[ANN05] Anne-Marie Déplanche, Sébastien Faucou Institut de Recherche en Communications et Cybernétique de Nantes (UMR no 6597), "Les langages de description d'architecture pour le temps réel".

[ARN07] Arnaud Cuccuru – Chokri Mraidha – François Terrier –Sébastien Gérard, "Méta-modèles et Points de Variation Sémantique". (SéMo'07) (Atelier adossé à la conférence francophone IDM'07 Toulouse les 29 et 30 mars 2007).

[ATL05] ATLAS group LINA & INRIA Nantes, " ATL: Atlas Transformation Language ATL". Starter's Guide version 0.1 December 2005

[BEN06] Benoit Combemale Sylvain Rougemaille, Xavier Crégut, Fedéric Migeon Marc Pantel Christine Maurel, "Expérience pour décrire la sémantique en Ingénierie des modèles". IDM6 LILE 26 28 juin 2006

[BEZ03] Bézevin Jean, Erwan Breton, Grégoire Dupé, Patricx Valduriez, "The ATL Transformation-based Model Management Framework", RESEARCH REPORT No 03.08 09/09/2003

[BUR90] A. Burns and A Wellings., "Real-Time Systems and their Programming Languages. Addison-Wesley", 1990.

[CLL73] C.L. Liu & J.W. Layland, "Scheduling algorithms for multiprogramming in a hard real-time environment",Journal of the Association for Computing Machinery 20 (1973), no. 1, p. 46-61

[CNR88] G.D.R.T.R CNRS. "Le temps réel. Technique et Science Informatiques", 1988.

[CWM01] "The Common Warehouse MetaModel (CWM), OMG Document ad/2001-02-01", Janvier 2001.

[DAV06] Dave Thomas Claude Baron Bertrannd Tondu. "Ingénierie dirigée par les modèles appliquée à la conception d'un contrôleur de robot de service". IDM6 LILE 26 28 juin 2006

[DEL03] Jérôme DELATOUR 2003. "Contribution à la spécification des systèmes Temps Réel L'approche UML/PNO" THÈSE Présentée au Laboratoire d'Analyse et d'Architecture des Systèmes (LAAS) Du CNR par Jérôme DELATOUR 2003

[DOU98] B. Douglass. "Real-Time UML: Developing Efficient Objects for Embedded Systems. Addison-Wesley", 1998.

[DUB05] Dubois Hubert, Gérard Sébastien, Mraidha Chokri. "Un Langage d'Action pour le développement UML de systèmes embarqués temps réel". CEA-List CEA Saclay 91191 Gif-sur-Yvette Cedex France, IDM'05 Premières Journées sur l'Ingénierie Dirigée par les Modèles Paris, 30 juin, 1 Juillet 2005.

[FOW97] FOWLER Martin. "Analysis patterns, reusable object models". 07-1997

[FRA04] Franck Chauvel DEA d'Informatique Sous la direction de M. Jean-Marc Jézéquel Rennes, "Génération de code à partir de modèles UML Avec points de variation sémantique". le 18 juin 2004 Université de RENNES 1 (IFSIC)

[GAS06] "Gaspard Profile". DART team Laboratoire d'informatique fondamentale de Lille.Université des sciences et technologies de Lille. France 2006.

[HAN95] C. Hanen and A. Munier. "Cyclic scheduling on parallel processors : An Overview, volume Scheduling theory and its applications", P. Chretienne et al., Chap 9. John Wiley & Sons, 1995.

[IME05] Imène Benkermi, Amine Benkhelifa, Daniel Chillet, Sébastien Pillement, Jean-Christophe Prevotet, François Verdier. "Modélisation niveau système de SoC reconfigurables". RENPAR'16 / CFSE'4 / SympAAA'2005 / Journées Composants Le Croisic, France, 5 au 8 avril 2005

[KMM05] "The Kernel Meta-MetModel (KM3) Manual", disponible dans le projet GMT 61, section ATL Documentation", Aout 2005.

[LUI03] Luís Gomes, Anikó Costa. "From Use Cases to System Implementation: Statechart Based Co-design". Proceedings of the First ACM and IEEE International Conference on Formal Methods and Models for Co-Design (MEMOCODE'03). ISBN 0-7695-1923-7/03 2003 IEEE.

[MAR06] "MARTE: the future OMG standard for MDE of RTES 1stworkshop on UML and AADL". ENST, Paris –October, the 9th2006

[MAR06] Maria Cruz Valiente, Gonzalo Genova, Jesus Carretero. "UML 2.0 Notation for Modeling Real Time Task Scheduling". Carlos III University of Madrid JOURNAL OF OBJECT TECHNOLOGY Published by ETH Zurich, Chair of Software Engineering ©JOT, 2006

[MOF03] "QVT, MOF 2.0 Query / Views / Transformations RFP", OMG Document ad/ 2003-08-03, Août 2003.

[NIZ06] Nizar Idoudi, Claude Duvallet, Bruno Sadeg, Faiez Gargouri. "Vers une méthode de conception des bases de données temps réel". GEI 2006

[OCL04] Eric Cariou. "OCL Object Constraint Language".Département Informatique Université de Pau. http://web.univ-pau.fr/~ecariou/cours/mde/cours-ocl.pdf

[OME05] Iulian Ober, Ileana Ober, Susanne Graf et David Lesens VERIMAG. "Projet Omega : Un profil UML et un outil pour la modélisation et la validation de systèmes temps réel". Grenoble Université Paul Sabatier, Toulouse (IRIT) EADS SPACE Transportation

[PAI06] Stéphane PAILLER. "Analyse Hors Ligne d'Ordonnançabilité d' Applications Temps Réel comportant des Tâches Conditionnelles et Sporadiques". THÈSE Présentée au École Nationale Supérieure de Mécanique et d'Aérotechnique le 19 Octobre 2006

[PET62] "C.A. Petri. Kommunikation mit automaten". Bonn Institut für Instrumentelle Mathematik, Schriften des IIM Nr. 2, English translation, 1966, pages Vol.1, Suppl.1, 1962.

[PIN04] G. Pinter and I. Majzik. "Impact of Statechart Implementation Techniques on the Effectiveness of Fault Detection Mechanisms". Proceedings of the 30th EUROMICRO Conference (EUROMICRO'04). 1089-6503/04 IEEE

[PRI04] Prih Hastono, Stephan Klaus and Sorin A. Huss. "Real-Time Operating System Services for Realistic SystemC Simulation Models of Embedded System". Integrated Circuits and Systems Laboratory Department of Computer Science - Technische Universität Darmstadt Alexanderstr. 10, 64283 Darmstadt, Germany

[QOS04] "UMLTM Profile for Modeling Quality of Service and Fault Tolerance Characteristics and Mechanisms". OMG Adopted Specification. Ptc/2004-06-01

[QVT03] "OMG / MOF 2.0, Query / Views / Transformation. ad/2002-04-10, Revised Submission", Version 1.0, 2003/08/18, OpenQVT, disponible à http://www.omg.org/docs/ad/03-08-05.pdf.

[REAL03] "Real-Time Concepts for Embedded Systems". CMP Books © 2003

[ROO96] "Tutorial: real-time object-oriented modeling (ROOM) Selic", B. Real-Time Technology and Applications Symposium, 1996. Proceedings, 1996 IEEE Volume, Issue, 10-12 Jun 1996 Page(s):214 – 217

[ROP04] Vincent ENGLEBERT. "Modélisations de systèmes coopératifs mobiles à temps réels Analyse de cas pour des systèmes ATC (Air Traffic Control) Evaluation du processus de développement ROPES". Thèse soutenue 2004

[SAM06] Samuel Rouxel. "Modélisation et Caractérisation de Plates-Formes SoC Hétérogènes : Application à la Radio Logicielle". Thèse présentée et soutenue publiquement le 5 décembre 2006 par

[SDL04] EMMANUEL GAUDIN PRAGMADEV. "SDL et UML: mariage de raison pour la conception des logiciels temps réel". Mars 2004 n°145 - Electronique

[SHO04] Shourong Lu Wolfgmg A. Halang Roman Gumzej. "Towards Platform Independent Models of Real Time Operating Systems". 0-7803-8513-6/04/$20.00 Q2004 IEEE

[SPT02] "UMLTM Profile for Schedulability, Performance, and Time Specification". An Adopted Specification of the Object Management Group, Inc. January 2005 Version 1.1 formal/05-01-02

[STA88] J.A. Stankovic. "Misconception about real -time computing". In IEEE Computer Magazine, volume 10, pages 0–19. 21, 1988.

[STE04] Stephan Flake and Wolfgang Mueller C-LAB. "An OCL Extension for Real-Time Constraints", , Paderborn University, F¨urstenallee 11 33102 Paderborn, Germany

[UML01] Unified Modeling Language (UML), OMG Document formal/200l-09-67, Septembre 2001.

[UML04] "Real Time UML: Advances in The UML for Real-Time Systems", Third Edition Pub Date February 20, 2004

[VIV02] Vivek Agarw. M.Tech . "Embedded Operating Systems for Real-Time Applications". Sagar P M (02307406). credit seminar report,Electronic Systems Group, EE Dept, IIT Bombay, Submitted in November 2002.

[XML01] "XML Metadata Interchange (XMl) ", OMG Document formal/2000-11-02, Novembre 2001.

[YVO05] Yvon Trinquet IRCCyN. "Les systèmes d'exploitation temps réel". – UMR CNRS 6597 Ecole Centrale de Nantes Université de Nantes. Ecole d'été Temps réel 2005 ETR'05