UML et le paradigme synchrone : Application à la conception de contrôleurs embarqués

Marie-Agnès PERALDI-FRATI, Charles ANDRÉ, Jean-Paul RIGAULT

Email : { map, andre, jpr }@unice.fr

Laboratoire I3S – UNSA/CNRS – UMR 6070

BP 121

06903 Sophia Antipolis cédex

Résumé : Nous proposons une approche qui combine les concepts des langages synchrones et d'UML pour la mise en œuvre de systèmes temps réel. Nous prenons un exemple de l'industrie automobile (un contrôleur de siège) pour illustrer notre approche. L'UML n'étant pas capable de prendre en compte certaines spécificités du synchrone, nous introduisons des modèles dynamiques et des stéréotypes mieux adaptés à une programmation réactive. La méthode proposée permet une conception à base d'objets qui prend en compte la sémantique des modèles synchrones et facilite la vérification formelle de propriétés. La réalisation des contrôleurs peut être totalement ou partiellement synchrone.

Mots clef : UML, Programmation synchrone, Contrôleur embarqué, Application automobile.

1 Introduction

1.1 Domaine d'application

Les systèmes autonomes intégrant un processeur numérique sont chaque jour plus nombreux. Certains sont impliqués dans des opérations de contrôle en temps réel. C'est le cas de contrôleurs que l'on trouve dans les automobiles. L'exemple qui illustre cet article a été emprunté à ce domaine d'application. Il s'agit d'un contrôleur pour siège de voiture « haut de gamme ». Cette application a fait l'objet d'un concours proposé à l'automne 2000 par Daimler-Chrysler[1]. Le but était de concevoir une application typique de contrôle dans le domaine automobile, avec une approche objet. Cette application présente des caractéristiques très réactives, les contraintes de temps réel ne sont pas très sévères, mais elle demande une attention particulière au niveau de la robustesse et de la prévisibilité du comportement. La réactivité signifie que le comportement du système résulte de ses interactions permanentes avec son environnement. La robustesse est une qualité qui permet au système d'être peu sensible à des informations erronées. La prévisibilité des comportements garantit que l'on sait en toute circonstance, comment va réagir le système. L'application étudiée possède, en plus, un caractère « embarqué ». Ce terme est ici tout à fait justifié et n'est pas une traduction approximative du mot « embedded ». Le système est soumis à des impératifs de consommation, de taille et de coût limités. Au niveau matériel, le cœur de la réalisation peut être un simple microcontrôleur. Ces contraintes induisent de nouvelles exigences d'efficacité, que ce soit au niveau du code ou au niveau des méthodes de conception.

L'approche que nous allons présenter a été appliquée à l'exemple ci-dessus, mais elle s'applique plus généralement aux **systèmes dits à contrôle prépondérant**, qui sont souvent des applications embarquées et temps réel. Lorsque les aspects transformationnels sont importants, par exemple dans des applications incluant du traitement du signal, la méthode nécessite des adaptations qui seront évoquées mais non détaillées. Il en est de même lorsque le système est hybride (à la fois de la commande par événements discrets et du contrôle continu).

Dans la suite, lorsque nous parlerons de contrôleur, il faudra essentiellement penser à systèmes à événements discrets à contrôle prépondérant.

1.2 Problèmes de conception de contrôleurs

Le contrôle a été longtemps abordé, surtout quand les cibles sont des microcontrôleurs, par des méthodes de bas niveau : langage assembleur, automates, grafcet.

Les solutions multitâches sur un RTOS font intervenir des langages de plus haut niveau, mais les programmes utilisent de nombreux mécanismes de synchronisation et de communication qui rendent difficiles leur mise au point et leur validation.

Les **langages synchrones** ont été introduits pour permettre une conception de haut niveau des systèmes réactifs. L'objectif initial était de produire du code sûr pour systèmes réactifs. Cet objectif est atteint par la simplicité des concepts sous-jacents et une définition mathématique de la sémantique de ces langages. Estérel [Ber00] et les syncCharts [And96] sont particulièrement intéressants pour les systèmes à contrôle prépondérant [ABD98]. Même si ces formalismes supportent la hiérarchie, la concurrence et la préemption, il n'est pas toujours aisé de structurer la solution. En fait, il y a encore à trouver une méthodologie adaptée.

---

[1] Voir le site http: //www.automotive-uml.com/mc/index.html

En génie logiciel, les **approches objets** sont maintenant largement utilisées : elles apportent des facilités de structuration et de meilleure ré-utilisation de solutions déjà éprouvées. Notre première idée a donc été d'aborder la conception de contrôleur en combinant l'approche objet et l'approche synchrone. L'utilisation d'UML dans ce cadre apparaît de nos jours comme incontournable pour mener à bien la modélisation objet. Toutefois, les contrôleurs ont des spécificités que les objets « classiques » capturent mal. Il fallait donc étendre la vision objet traditionnelle. La méthode ROOM [SGW94] contient les germes de telles extensions, dont la plupart ont des chances d'être adoptées dans les prochaines versions d'UML. Nous avons repris plusieurs idées de ROOM en les adaptant à l'approche synchrone. Nous avons également adapté UML lui-même à cette approche.

1.3 L'approche proposée

Dans cet article, nous montrons comment nous combinons les avantages d'UML (qui peut représenter à la fois des aspects structurels et comportementaux) et de la programmation synchrone. Il ne s'agit pas d'une simple juxtaposition. Nous désirons les **intégrer** dans une approche méthodologique de conception d'applications réactives sûres.

Ceci n'est d'ailleurs pas si simple. En effet, il a déjà été remarqué qu'une méthodologie adaptée au paradigme synchrone restait à définir. Et, contrairement à des méthodes de développement de logiciel classiques (comme SA/RT pour rester dans le domaine du temps réel), UML se veut indépendant du processus de développement. Aussi ce papier ne peut-il actuellement proposer une méthodologie complète et détaillée pour « UML synchrone », mais seulement tracer les grandes lignes des outils nécessaires, des modifications indispensables à UML et de l'enchaînement grossier des différentes activités (ce que nous appelons le « workflow »).

L'exemple choisi est essentiellement un problème de contrôle à événements discrets, les aspects transformationnels sont très limités. De plus, le contrôleur de cette application peut être un simple microcontrôleur. La solution peut donc être fortement centralisée.

Ces caractéristiques risquent de faire croire que les propositions que nous faisons n'ont qu'un usage limité à une classe réduite d'applications embarquées de petites tailles et nécessitant peu de calculs. La méthodologie proposée va bien au-delà.

1.4 Plan de la présentation

Dans la première section nous justifions notre approche en soulevant les problèmes liés à l'utilisation d'UML pour la modélisation de systèmes à événements discrets. Nous montrons l'apport des techniques synchrones dans ce domaine. Dans une deuxième section nous présentons au travers d'un « workflow » notre manière de fusionner les deux approches en prenant les points forts de chacune. Nous appliquons ce "workflow" à l'exemple d'un contrôleur de siège automobile en montrant en particulier comment les parties critiques du contrôleur peuvent être validées comme étant conformes aux spécifications. En conclusion nous discutons sur les bénéfices et les limites de l'approche. Finalement nous indiquons comment ces travaux peuvent déboucher sur la définition d'un « profile » temps réel synchrone pour UML.

2 UML, le temps réel et l'approche synchrone

2.1 UML et le temps réel

UML est un modèle universel, orienté-objets, pour l'analyse et la conception de systèmes. Son ambition est de couvrir tous les domaines d'application et de parvenir à « fondre » ces applications dans un moule objet. Il s'avère que si quelques modèles d'UML ont effectivement ce caractère universel, d'autres nécessitent une spécialisation liée à la classe d'application. Cette nécessité d'introduire des extensions est reconnue par la communauté UML et est rendue possible par la notion de « profil » (profile) qui vise à installer UML comme une famille de langages [Coo00].

Plusieurs profils ont déjà été proposés pour prendre en compte les caractéristiques des systèmes temps réel (voir par exemple [KN99] pour une liste non exhaustive). Deux d'entre eux sont assez populaires : RT-UML et UML-RT. *Real-Time UML* (RT-UML), décrit dans le livre de B.P. Douglass [Dou99], s'appuie sur UML pour modéliser les systèmes temps réel. RT-UML est une méthodologie fondée sur UML sans y ajouter d'extension autre que des diagrammes temporels. Les modèles sont simulables et exécutables. Un outil industriel est associé à RT-UML (Rhapsody de la société I-Logix). *UML for Real-Time* (UML-RT) [SR98] va un peu plus loin. C'est une adaptation de la méthode ROOM [SGW94]. Elle introduit un stéréotype de classes appelé *Capsule* ainsi que les notions de *Port*, *Protocole* et *Connecteur*, mieux adaptés aux systèmes temps réel que les classes. ROOM introduit également une nouvelle représentation, le *diagramme de structure*, qui permet de décrire la structure d'une agrégation d'objets et surtout l'interconnexion de ces objets. Ici encore les modèles sont simulables, exécutables. Une implémentation de UML-RT est proposée dans l'outil Rose Real Time de Rational. Bien évidemment, aucune de ces deux approches ne prend en compte le modèle synchrone.

D'autres travaux sur UML temps réel sont menés par l'Object Management Group (OMG). Un appel à propositions a été lancé (RFP 2.0) sur les thèmes de modèle de ressource, modèle du temps et modèle d'ordonnancement. Il a fait l'objet d'une seule réponse [OMG00], qui exclut explicitement toute prise en compte de notion de temps logique. Le même constat est applicable à UML-RT et RT-UML qui manipulent un temps discret (strictement périodique) et/ou continu mais lié au temps physique et non logique.

Nous avons néanmoins retenu de ces méthodes et modèles de très bonnes idées pour ensuite les adapter à la modélisation de systèmes discrets synchrones.

2.2 Le paradigme synchrone

Les langages synchrones ont été introduits pour traiter le problème de la programmation réactive. Le terme *synchrone* correspond à la sémantique d'exécution des programmes. Un système synchrone peut être vu comme une boîte noire qui reçoit des entrées de l'environnement et génère des sorties vers cet environnement. Les interactions se font à des instants discrets ce qui donne un sens non ambigu à la simultanéité d'occurrences de signaux. Les réactions du système se font en *temps nul* et de *manière synchrone* avec les stimuli qui les ont déclenchées.

Les programmes synchrones considèrent un temps logique et multiforme. Ceci signifie que n'importe quel signal répétitif peut être utilisé comme base de temps. Une autre caractéristique est la communication qui est supportée exclusivement par les signaux qui peuvent véhiculer des valeurs. Les signaux sont diffusés de manière instantanée autorisant à la fois des communications *1-to-many* mais également *many-to-many*. Les valeurs portées par ces signaux peuvent être combinées à l'aide de fonctions associatives et commutatives

associées aux signaux lors de leurs déclarations. Ce type de comportement est courant pour le matériel (par exemple des bus travaillant en « OU-cablé »).

Grâce à l'abstraction des interactions temps réel qu'apportent ces hypothèses fortes, il a été possible de donner aux langages synchrones des sémantiques mathématiques simples et expressives. Ces fondements mathématiques facilitent l'usage de techniques formelles pour la validation de programmes synchrones. Le modèle sous jacent aux langages synchrones est le modèle état-transition. Un programme synchrone peut être compilé en un système fini d'équations booléennes qui définissent le contrôle et déclenchent des actions. Ce système d'équations peut très bien être implémenté sous forme de code séquentiel complètement déterministe.

Parmi les langages synchrones nous avons retenu le langage impératif Esterel. Ce langage convient très bien pour les systèmes à contrôle prépondérant. Le langage apporte par ses constructions réactives, une structuration claire du contrôle. Lorsque les états sont explicitement utilisés, il est préférable de choisir le modèle graphique associé à Estérel et appelé « SyncCharts » [And96]. Ce modèle manipule macro-états et transitions entre états. Il est inspiré des Statecharts [Har87]. Les deux modèles diffèrent toutefois sur leur sémantique et leur expressivité. La sémantique mathématique des SyncCharts est pleinement compatible avec celle du langage Estérel. Les SyncCharts offrent, vis à vis des Statecharts, une plus grande richesse au niveau de l'expression des préemptions. La totale compatibilité entre Esterel et SyncCharts fait que ces deux modèles peuvent être utilisés conjointement. La plate-forme commerciale « Esterel-Studio » [ES] exploite cette possibilité en offrant éditeurs, compilateurs, simulateurs pour ces deux modèles, ainsi qu'un outil de validation dédié (XEVE, [Bou98]).

Actuellement l'approche synchrone supporte de nombreux outils et notations mais pas de méthodologie. De plus ce n'est pas une approche orientée objets bien qu'elle puisse être compatible avec les concepts orientés objets. Dans ses travaux de thèse, F. Boulanger [Bou93] a introduit les objets synchrones. L'idée est d'associer une classe à un programme synchrone. Les signaux du programme sont considérés comme des méthodes de cette classe et une méthode particulière (activate) exécute les réactions. Il est possible de faire coopérer de façon synchrone plusieurs objets synchrones. Les résultats les plus significatifs ont été obtenus pour des compositions statiques d'objets synchrones. Un prolongement de ces travaux est présenté dans [ABPRV96] et exploite les concepts d'OMT [RBP91], un des précurseurs d'UML. Le but est de définir *les objets réactifs synchrones* ainsi que leur composition. L'utilisation d'OMT est limitée à la description de la structure des classes et leurs associations, les SyncCharts sont utilisés pour exprimer les comportements.

2.3 UML et le paradigme synchrone

Rapprocher UML et le paradigme synchrone est apparu récemment comme un problème de recherche.

Parmi les premiers travaux, Dassault Aviation [BNLD00] propose une intégration des modèles synchrones et d'UML. Ils définissent des extensions objets du langage Esterel (Esterel++). Ils utilisent les SyncCharts pour modéliser le comportement dynamique des objets en empruntant à ROOM/UML-RT les notions de capsules, protocoles, … qui deviennent des classes particulières mais avec une sémantique mieux adaptée au modèle synchrone. Ces travaux adoptent également les diagrammes de structure de ROOM/UML-RT. Ces notions ainsi qu'une possibilité de génération de code Esterel++ sont intégrées dans l'environnement Rational Rose. Comme Rational Rose ne permet pas l'édition de SyncCharts, les auteurs de cette méthode utilisent l'environnement Esterel Studio [ES]. La combinaison des deux environnements est maintenant adoptée chez Dassault Aviation.

Toujours dans le but de rapprocher UML et le synchrone, nous avons récemment présenté [APR01] la possibilité d'exprimer des scénarios en prenant en compte les aspects synchrones (simultanéité des événements, diffusion instantanée, préemption…). Ce nouveau modèle appelé SIB (*Synchronous Interface Behavior)* est une extension des diagrammes de séquence et s'inspire des *Message Sequence Charts* (MSC) [RGG96]. Les SIB

peuvent évidemment être traduits en Esterel. Cette possibilité est nouvelle par rapport à Esterel++ [BNLD00]. Une autre évolution par rapport à l'approche de Dassault Aviation est la définition d'ilôts synchrones. Ce concept permet des déploiements *globalement asynchrones et localement synchrones* (les îlots étant associés aux parties localement synchrones).

Dans le domaine de la programmation synchrone déclarative, des travaux ont été entrepris à l'IRISA. Un formalisme synchrone de pré-ordres étiquetés, appelé BDL (Behavioral Description Language), a été introduit. Les Statecharts et les diagrammes de séquences d'UML sont traduisibles en BDL [WTBL00]. Cette transformation autorise, ensuite, des manipulations formelles.

Une difficulté, mise en évidence par ces travaux, résulte de la faiblesse sémantique d'UML. Le comportement dynamique des objets en UML est généralement représenté par les Statecharts et leurs interactions sont exprimées par les scénarios textuels et les diagrammes de séquences. De nombreuses règles (syntaxe abstraite) permettent de s'assurer que les modèles sont « bien formés », mais au niveau des comportements, il reste beaucoup à faire. Certes, les diagrammes d'états-transitions (les Statecharts adoptés par UML) ne sont pas le modèle d'UML le plus dénué de sémantique. Toutefois, leur sémantique actuelle est insuffisante pour représenter correctement le comportement synchrone. Par ailleurs la relation entre les différents types de diagrammes n'est pas toujours correctement formalisée en UML. Ainsi, si la liaison entre diagrammes de séquences et diagrammes de classes est (partiellement) supportée par la plupart des outils, c'est loin d'être le cas pour la liaison entre le modèle de classes et les Statecharts[2] ou encore pour la relation entre diagrammes de séquence et Statecharts. Or, pour les applications que nous envisageons, cette cohérence entre structure statique et comportement réactif ou encore entre les différents modèles de représentation du comportement, est le nœud du problème. Un dernier point de divergence à noter est que les hypothèses faites dans la sémantique donnée par UML à ses machines d'état [OMG99, Chapitre 2.12] n'autorisent pas les communications par combine/broadcast des langages synchrones.

En substituant les SIB aux diagrammes de séquences et les SyncCharts aux Statecharts nous conservons la démarche d'UML mais nous ajoutons les bénéfices de l'approche synchrone. Le bénéfice de cette approche est de disposer de modèles cohérents car issus de la même sémantique synchrone. Ils peuvent de ce fait être traduits dans le même langage exécutable (Esterel), et ainsi hériter des plates-formes déjà existantes.

La section suivante explique comment ces modèles, qui ne sont pas ceux proposés dans l'UML, peuvent être mis en œuvre pour développer des solutions synchrones, tout en respectant la philosophie d'UML. Le travail qui consiste à intégrer formellement ces nouvelles représentations dans UML (c'est-à-dire dans le *méta-modèle* d'UML) est en cours mais ne sera pas décrit cet article. Qu'il suffise de savoir qu'il nécessite sans doute de recourir à des extensions « lourdes » d'UML.

3 UML synchrone

3.1 L'approche

Notre but est d'apporter aux techniques synchrones des modèles tels que ceux d'UML et UML-RT et qui permettent la modélisation fonctionnelle (cas d'utilisation ou use cases), la modélisation objet (diagramme de classes), la modélisation de l'architecture (diagrammes de structure). Ainsi, comme le montre la Figure 1, nous proposons un « workflow » qui intègre des modèles d'UML et d'UML-RT. Certains modèles ont été adaptés

[2] Bien que ROOM et UML-RT améliorent considérablement la situation dans ce domaine.

pour prendre en compte les caractéristiques des systèmes synchrones, d'autres sont carrément remplacés par de véritables modèles synchrones (SyncCharts, SIB).

Notre démarche a également un deuxième objectif qui est de fixer et formaliser les relations entre les instances de Cas d'Utilisation et les modèles d'expression du comportement. C'est à ces fins qu'ont été introduits les SIB. Notre volonté est de proposer une meilleure compatibilité entre modèles dynamiques, permettant ainsi d'établir formellement certaines propriétés.

Nous détaillons dans la suite du document la démarche « UML synchrone » en précisant les modifications apportées par rapport à une approche classique UML.

- *Modèle des Cas d'Utilisation* : il apporte une vue fonctionnelle hiérarchique de l'application. Nous utilisons ici le modèle standard proposé par UML.
- *Modèle d'objets et modèle de classes* : ils précisent les objets, attributs, opérations et les associations qui les lient. Nous avons dû spécialiser ces modèles. Nous avons recours à des stéréotypes dont les *S_Capsule*, *Islet, Pluget et Connection.* Ces stéréotypes sont proches de stéréotypes introduits dans UML-RT (capsule, protocol, connector).
- *Réalisation des Cas d'Utilisation* : Par les « collaborations UML », nous précisons quelles sont les classes impliquées dans la réalisation d'un cas d'utilisation.
- *Diagramme de structure* : Ce modèle est une extension synchrone des diagrammes de structures que l'on rencontre en UML-RT. Il exprime la composition structurelle des *S_Capsules* pour former des *S_Capsules* de plus haut niveau ou bien des *Islets* (cas de la composition synchrone).
- *Modèles Dynamiques :* Nous avons été amenés à remplacer le diagramme de séquence d'UML par les SIB (Synchronous Interface Behavior), le modèle d'état d'UML par les SyncCharts.

UML ne traite pas directement les questions de validation, alors qu'il s'agit d'un point fort de l'approche synchrone. Notre workflow intègre donc ces possibilités. Des propriétés peuvent être exprimées sous forme de SIB ou de SyncCharts. Nous pouvons ainsi exprimer le comportement et les propriétés attendus des contrôleurs avec des modèles (SIB, SyncCharts ou Esterel) qui ont des sémantiques pleinement compatibles. De plus, comme dans Rose RT ou Rhapsody, nos modèles sont exécutables et simulables. Dans notre cas, nous nous appuyons sur l'environnement Esterel pour la simulation, la génération de code, et la connexion à l'outil de validation Xeve.

3.2 Mise en œuvre

Nous illustrons notre approche en l'appliquant à un exemple industriel qui nous a été fourni par Daimler Chrysler dans le cadre d'un concours lancé auprès d'industriels et de laboratoires de recherche.

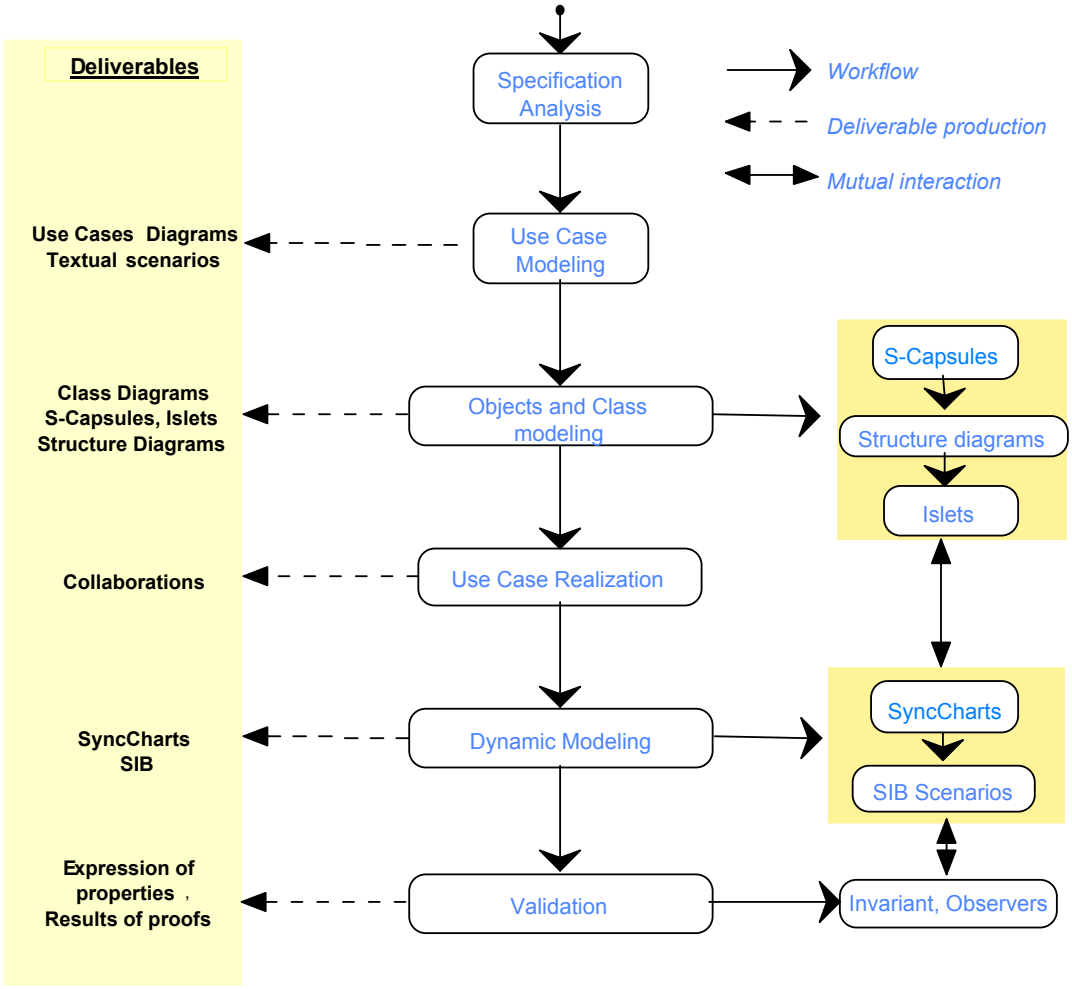


Figure 1: Le workflow de UML synchrone.

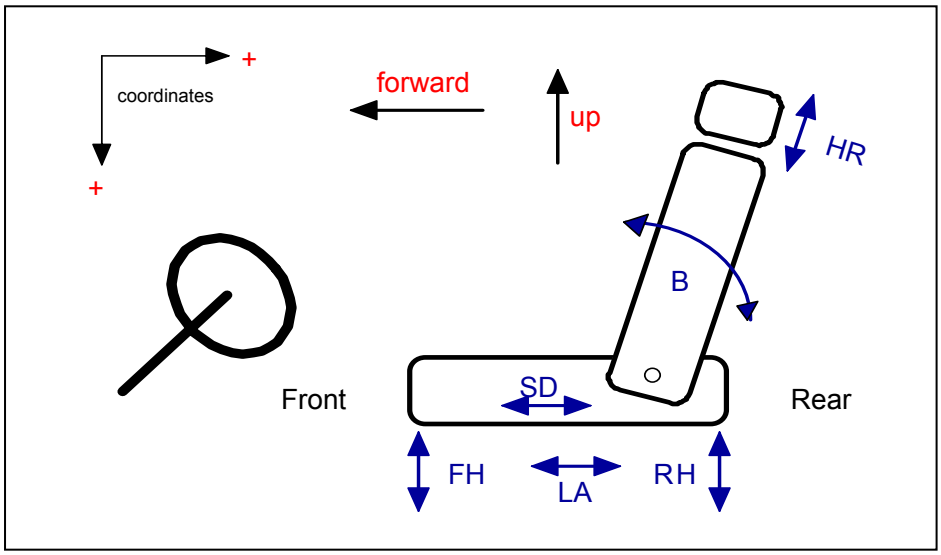
4 Étude de cas : contrôle d'un siège automobile

4.1 Le siège

D'un point de vue physique, le siège est constitué de 6 moteurs qui permettent la modification de la position de ses différentes parties (distance au volant, profondeur de l'assise, inclinaison du dossier, hauteur de l'appui-tête…). Un moteur a 3 états ; il peut : être stoppé, tourner dans un sens ou dans l'autre. Lorsqu'un moteur tourne, il émet un « tic » (signal tick) à chaque révolution. Les ajustements sont limités en amplitude et caractérisés par leur nombre de tics. La

Figure 2 donne une vue des différents mouvements ainsi que les noms des moteurs qui réalisent ces déplacements. Les valeurs numériques ont été volontairement omises car elles relèvent de spécifications non publiques.

Il y a des priorités à respecter pour l'activation de moteurs. Ils existent 2 groupes de 3 moteurs : LA (Longitudinal Adjust), RH (Rear Height), SD (Squab Depth) d'une part et B (Backrest), FH (Front Height), HR (Head Restraint) d'autre part. Les noms des moteurs sont indiqués par ordre de priorité. Un seul moteur peut être actif à l'intérieur d'un groupe ce qui signifie en particulier qu'un moteur doit être stoppé si que commande arrive pour un moteur plus prioritaire. En revanche 2 moteurs appartenant à 2 groupes différents peuvent être actifs simultanément.



L'activation des moteurs se fait par un panneau de contrôle que manipule le conducteur du véhicule, c'est ce que nous appellerons *ajustement manuel*. A cette fonctionnalité s'ajoutent des *ajustements automatiques* qui peuvent être déclenchés en cas de *recalibrage* de modes *courtoisie* ou de *mémorisation*. Ces déclenchements ont des origines variées (expiration de temporisations, ouverture de la porte, restauration d'un état mémorisé du siège …). Enfin des fonctionnalités de *chauffage* et de *protection* mettent en jeu des temporisations et des mécanismes de préemption.

Dans la suite, nous notons M (Minus), Z (Zero), P (Plus) les trois commandes applicables à chaque moteur.

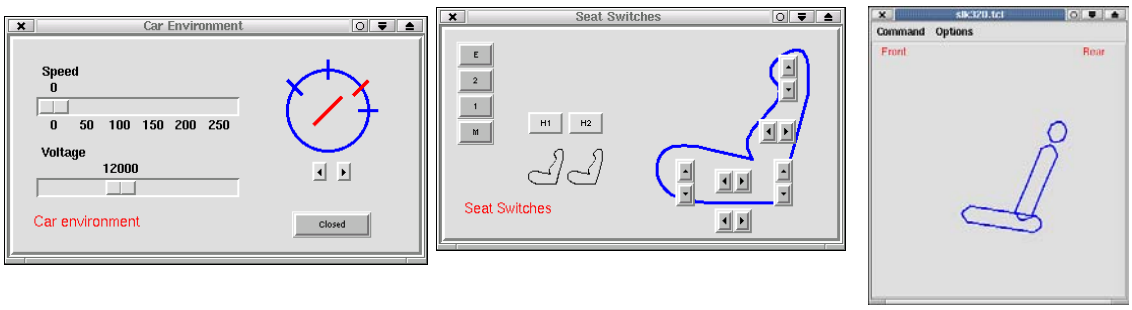
4.2 La plate-forme

La plate-forme utilisée pour réaliser le contrôleur de ce siège comporte des outils de conception des modèles, de génération de code, de simulation et de validation.

Pour la modélisation et la programmation nous utilisons la plate-forme Esterel Studio qui intègre les SyncCharts, Esterel ainsi que les compilateurs qui génèrent à partir de ces modèles des automates codés en C. Nous utilisons le simulateur XES pour simuler ces modèles. Sur ce simulateur on visualise les entrées et sorties du système. Dans cet exemple une telle simulation ne nous satisfaisait pas complètement, aussi avons nous

développé un simulateur en TCL/TK qui donne une vue plus réaliste des déplacements du siège. Les différentes parties de ce simulateur sont représentées sur la Figure 3.

Ce simulateur est inter opérable avec le contrôleur car tous les deux utilisent la même interface (celle imposée par les spécifications). La partie (a) du simulateur correspond à l'environnement du véhicule, la partie (b) au panneau de contrôle mis à disposition du conducteur et enfin la partie (c) affiche en temps réel les déplacements du siège. La réalisation tourne sur une machine Linux qui exécute deux processus : le simulateur Tcl/Tk du siège et le contrôleur que nous avons développé. Les communications se font via des sockets.



(a) **(b)** **(c)**

Figure 3 Simulateur du siège

4.3 Les aspects fonctionnels

Nous appliquons maintenant le workflow synchrone à la conception du contrôleur. Nous donnerons des exemples dans chacun des modèles et en particulier ceux dans lesquels apparaît l'apport du synchrone.

4.3.1 Les modèles

La Figure 4 (a) montre le premier niveau de cas d'utilisation. Les acteurs du système sont :

- Le panneau de contrôle
- La clef de contact
- Le siège, les moteurs, les capteurs …
- L'alimentation électrique
- Le tachymètre
- La portière

Les acteurs sont en interaction avec le système représenté par ses cas d'utilisation. Nous avons identifié 3 cas d'utilisation: Ajustement, Chauffage et Protection. Ajustement est raffiné sur la Figure 4 (b). On y retrouve les fonctionnalités indiquées dans la présentation du siège.

Pour préciser un peu plus les cas d'utilisation nous élaborons des scénarios dans un format textuel. Ces scénarios sont issus des spécifications et doivent comporter tout ou partie des champs suivants : **Use case name**, **Scenario name**, **Precondition**, **Steps, Postcondition**.

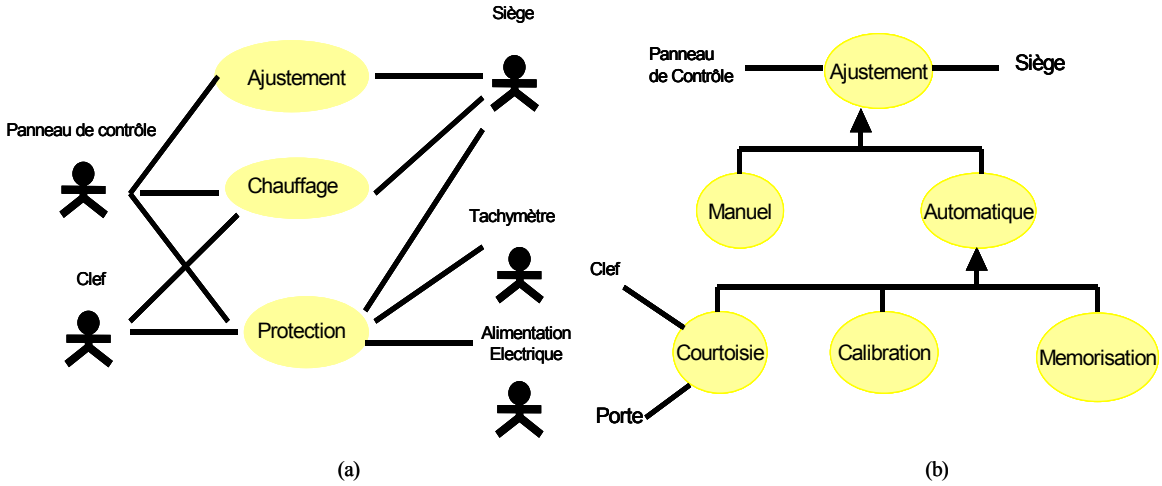


Figure 4 Cas d'Utilisation

4.3.2 Point de vue méthodologique

Cette phase de description fonctionnelle est essentielle dans les applications qui nous concernent. Durant cette phase nous essayons d'utiliser un « guide de conception » issu de notre expérience de modélisation mais il n'y a pas d'innovation particulière.

4.4 Les aspects structurels

A partir des cas d'utilisation nous définissons le modèle de classes, de collaboration et de structures de l'application.

4.4.1 Diagramme de classes

La Figure 5 est le diagramme de classes de plus haut niveau de notre solution. Sur cette figure apparaissent les notions de *s_capsules* et d'*islets*, et des *plugets*. Le méta-modèle qui définit ces éléments de modèle sort du cadre de cet article. Nous nous contentons de les présenter informellement et de montrer leur utilisation sur l'exemple.

Les capsules ont été introduites dans UML-RT comme des stéréotypes des classes pour prendre en compte les aspects contrôle des applications temps réel, avec en particulier la notion de communication par ports, qui eux-mêmes portent des signaux. Nous spécialisons le stéréotype capsule en « s_capsule » (Synchronized_Capsule) en y ajoutant un port spécial (clockPort) qui permet son contrôle.

Une « s_capsule » est donc une classe qui communique avec l'extérieur (dans les deux sens) uniquement par des *ports* (« Port »). A un *Port*, on associe des signaux (« s_signal »), qui peuvent être *input* ou *output*. Un *port* est typé par un « pluget ». Les *plugets* jouent un rôle d'interface et par leur définition récursive permettent une structuration hiérarchique des *ports*. Le « pluget » est très proche du « protocol » d'Esterel++, lui-même inspiré du « protocol » d'UML-RT. Les *plugets* n'étant pas strictement équivalents aux *protocols,* nous avons choisi un nom différent pour éviter toute confusion.

A partir de l'exemple nous précisons ces notions.

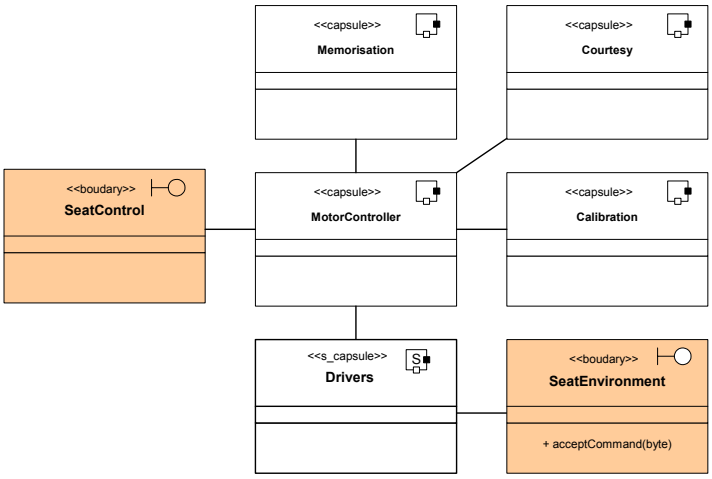


Figure 5 Diagramme de classe de l'application (haut niveau)

Les classes SeatEnvironment et SeatControl (Figure 5) dont le stéréotype est *boundary* sont les interfaces imposées par la spécification. Les autres classes sont celles définies pour réaliser le contrôle. Seule la *s_capsule* Drivers est détaillée par la suite.

La Figure 6 représente le diagramme de classe relatif au contrôle des moteurs. La *s_capsule* Drivers est une composition d'une *s_capsule* Encoder et de 2 *s_capsules* Group eux-mêmes étant composés de 3 MotorDriver. Les groupes sont ceux définis dans la spécification du siège.

Le compartiment inférieur des *s_capsules* précise les *ports*. Les *ports* peuvent être orientés en entrée (*i_port),* en sortie (*o_port*) ou mixtes (*port*). Ils sont typés par les *plugets.* Les ports non explicitement typés (e.g., ack de la *s_capsule* MotorDriver) contiennent un seul signal pur au sens du langage Estérel.

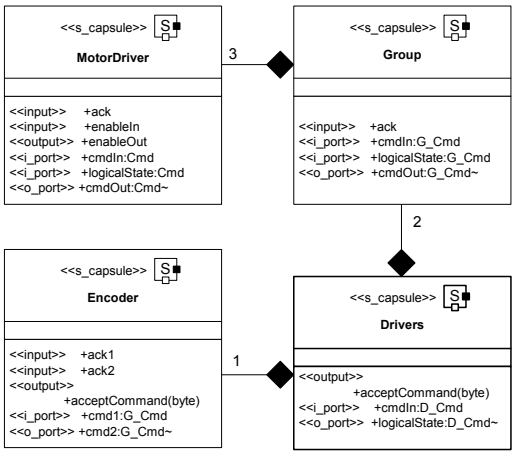


Figure 6 : Diagramme de classe du contrôle bas niveau des moteurs

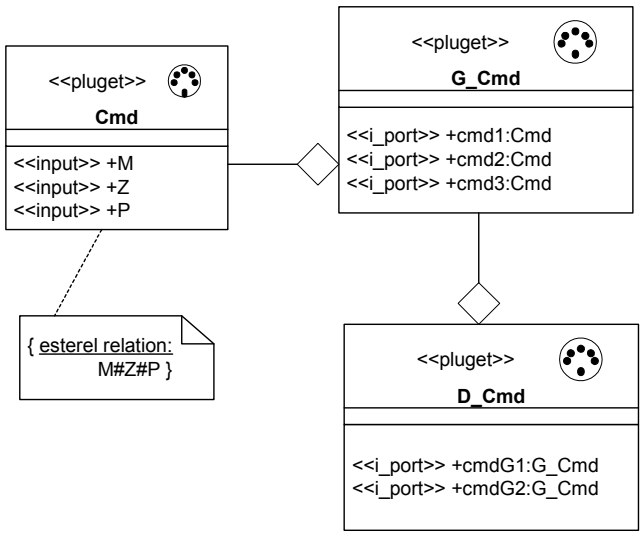


Figure 7 <<Plugets>> utilisés dans le contrôleur

La Figure 7 précise les *plugets* utilisés dans l'application. Le *pluget* Cmd est une interface constituée de 3 signaux « input » nommés respectivement M, Z, P. Une contrainte est attachée, précisant que ces 3 signaux sont exclusifs c'est-à-dire que ces signaux ne peuvent pas être présents simultanément. Il peut sembler étonnant de devoir exprimer une relation d'exclusion entre signaux (qui est le modèle adopté par RT-UML). Cela vient du fait qu'en UML « synchrone » c'est la simultanéité des signaux qui constitue le cas «classique ». Cela vient des hypothèses issues des modèles synchrones.

On voit sur la Figure 7 que le *pluget* G_Cmd est constitué de 3 « i_port » (Port d'entrée) qui sont eux-mêmes typés par le *pluget* Cmd. On peut comme cela réaliser une hiérarchie de *plugets* et alléger ainsi la représentation des *s_capsules* qui peuvent être, elles-mêmes, issues d'une composition hiérarchique d'autres *s_capsules*.

Notons qu'à tout *pluget* on peut associer le *pluget* conjugué noté par le même nom, mais avec le symbole ~ comme suffixe. Il correspond à la même interface, mais avec inversion des directions (un *input* devient *output*, un *i_port* devient *o_port*, et vice-versa).

4.4.2 Collaborations

Une fois élaboré le diagramme de classes on peut représenter les **collaborations**, c'est à dire les classes impliquées dans la réalisation des cas d'utilisation. Sur la Figure 8 nous prenons l'exemple de l'ajustement manuel.

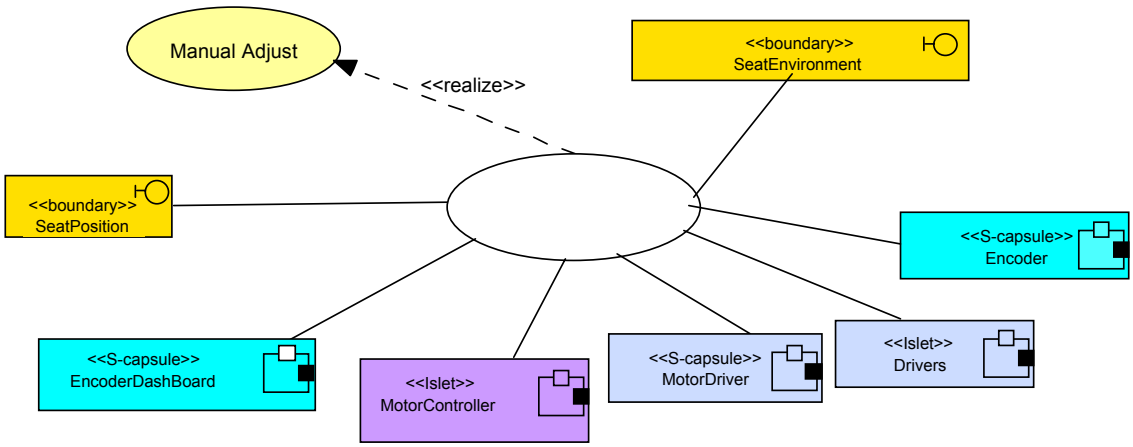


Figure 8 Collaboration pour le calibrage (Calibration)

4.4.3 Diagrammes de structure

L'étape suivante consiste à préciser chaque collaboration par des **diagrammes de structure**. Ce diagramme décrit les connexions entre *ports*. La Figure 9 représente la *s_capsule* MotorDriver. Les ports, leurs directions et leurs types sont explicités sur la figure. Noter que le clockPort d'une *s_capsule* est implicite.

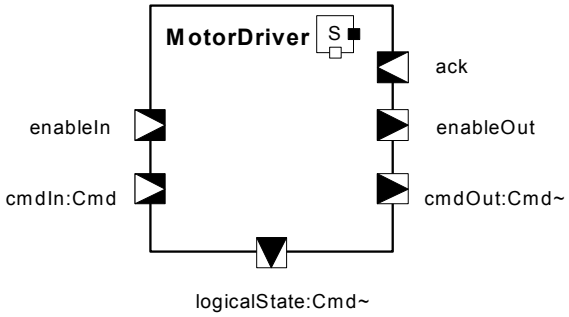


Figure 9 « s_capsule » du MotorDriver

Les *capsules* sont ensuite composées pour fournir des *capsules* plus complexes (Figure 10 et Figure 11). Cette représentation est générique : les paramètres formels sont indiqués dans le cadre pointillé situé au coin supérieur droit. Ces figures définissent respectivement la structure des *s_capsules* Group et Drivers.

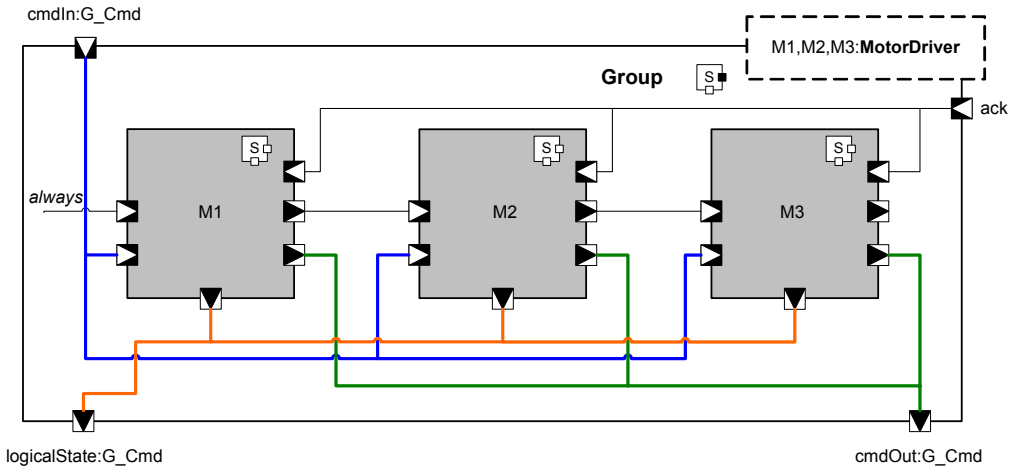


Figure 10 Diagramme de structure du Group.

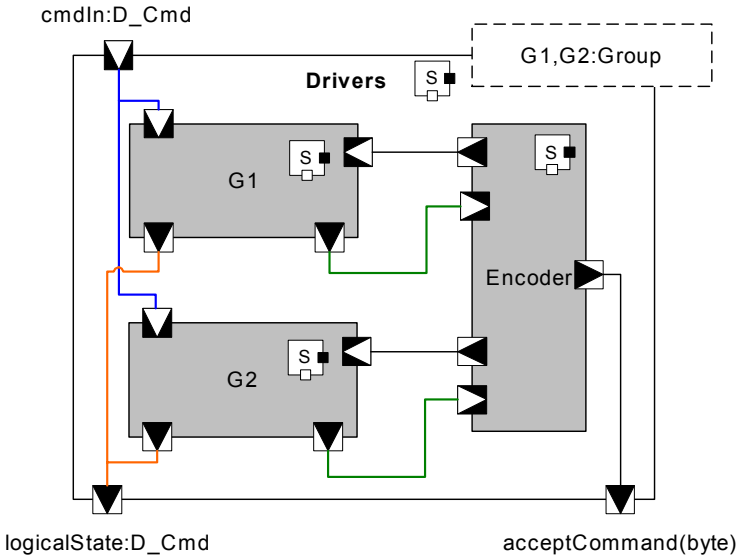


Figure 11 Diagramme de structure de Drivers

Lors du déploiement, le concepteur peut décider de regrouper des *s_capsules* au sein d'un même *islet* (un îlot synchrone). Un « islet » est une collection de *s_capsules* ; le clockPort de chaque *s_capsule* est relié à une capsule unique, propre à l'islet, qui détermine la notion d'instant pour l'*islet*. Le comportement émergeant de l'*islet* résulte de la composition synchrone des syncCharts qui expriment le comportement de chacune des *s_capsules*.

Dans notre application nous avons créé une instance *d'islet* appelée CoreControl définie par : Drivers(Group(LA,RH,SD), Group(B,FH,HR))

CoreControl est une implémentation synchrone du contrôle de bas niveau des moteurs.

4.4.4 Point de vue méthodologique :

Les diagrammes de classes et de structures fournissent une vision descendante de l'architecture du système. En revanche, les comportements se composent de manière ascendante. La composition est définie par le diagramme de structure. En choisissant les *islets*, on peut rendre certaines parties de l'application synchrone. Ce choix est lié à la criticité des parties en question et donc au besoin de prouver formellement des propriétés les concernant.

4.5 Les aspects comportementaux

4.5.1 Les modèles

En UML, les signaux déclenchent des changements d'états et peuvent provoquer des exécutions d'actions. Ces changements sont exprimés en UML par des Statecharts. Dans notre approche ce sont les SyncCharts, qui ont été préférés pour des raisons de cohérence sémantique avec nos modèles synchrones. La Figure 12 représente le SyncChart qui exprime le comportement de la *s_capsule* MotorDriver. MotorDriver reçoit des commandes (cmdIn : Cmd) et retransmet ou pas, selon son état, ces commandes en sortie (cmdOut : Cmd~). Il prend également en compte les possibilités de préemption des commandes et la gestion d'états transitoires.

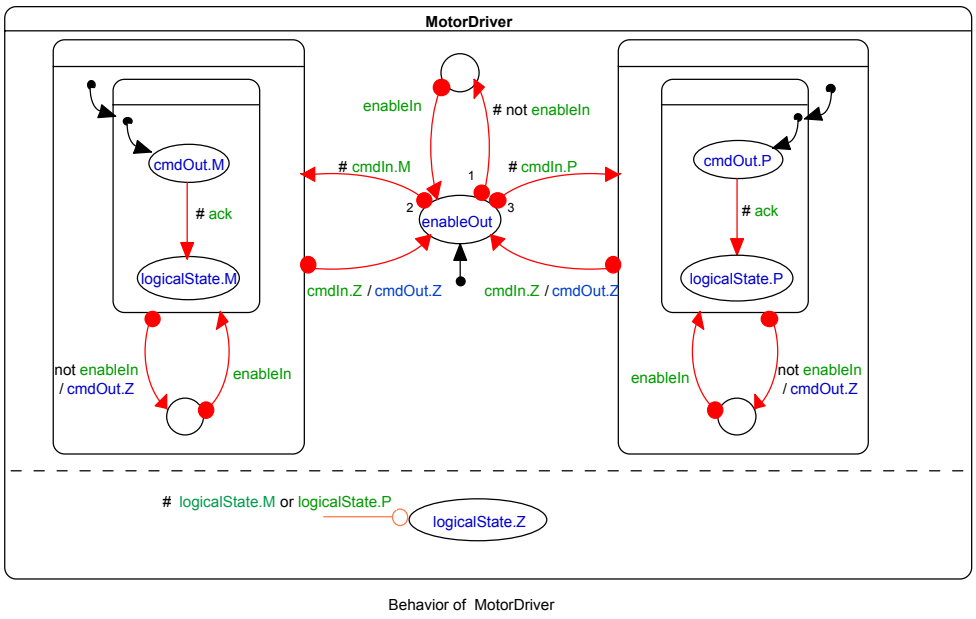


Figure 12 SyncChart de Motor Driver

Outre le comportement intra-objet, UML permet de spécifier les échanges dynamiques entre objets, dans le cadre des scénarios représentés par des diagrammes d'interaction (diagrammes de séquences ou de collaborations). Pour la même raison de cohérence avec le modèle synchrone, nous avons été conduits à substituer aux diagrammes de séquences, les SIB. La définition des SIB et leur application au contrôle du siège automobile ont été récemment publiés [APR01].

4.5.2 Point de vue méthodologique

SIB et SyncCharts sont des modèles simulables, mais aussi exécutables. Étant fondés sur une sémantique formelle commune, ils permettent de spécifier de manière cohérente les comportements intra et inter objets, et aussi ils facilitent la vérification formelle de cette cohérence.

Un autre point fort de notre approche, est la richesse d'expression des modèles comportementaux synchrones et leur facilité de composition, au moins en ce qui concerne les applications à contrôle prépondérant.

4.6 Validation

L'environnement de programmation synchrone assure la génération de code, possiblement instrumenté, ce qui facilite simulation et automatisation des preuves. Bien sûr, un minimum d'instrumentation et de développement d'interface est nécessaire.

4.6.1 Simulation

La simulation est une première étape indispensable dans le processus de validation. Elle permet de détecter des divergences par rapport au cahier des charges et de les corriger (modification du contrôleur ou des spécifications).

En termes méthodologiques, la simulation est un moyen de dialogue avec les commanditaires, facilitant un prototypage visuel.

4.6.2 Preuves

Pour les parties critiques de l'application, les propriétés ont été établies par vérification symbolique de modèles (symbolic model-checking) grâce à l'outil Xeve [Bou98]. Les propriétés de l'*islet* CoreCtrl ont été établies incrémentalement par *s_capsules* :

1. Pour Encoder : on prouve que l'ordre de priorité à l'intérieur des groupes est bien respecté. On établit également que les ordres d'arrêt des moteurs sont plus prioritaires que les ordres de marche.
2. Pour MotorDriver : on vérifie que les signaux M,Z,P sont bien exclusifs.
3. Pour Group : on montre que
   - Toute demande d'arrêt d'un moteur quelconque est immédiatement servie. 69202 états sont analysés pour cette propriété.
   - Les actions de suspension et de reprise des commandes de moteurs sont correctement effectuées.
4. Pour Drivers : on établit des propriétés de temps de réponse borné (bounded-responsiveness). Ces propriétés sont plus coûteuses en temps de calcul.

5 Conclusion

Nous combinons UML et l'approche synchrone afin d'apporter aux langages synchrones un cadre de conception orienté-objet. Pour cela il faut réaliser des extensions aux modèles d'UML pour qu'ils prennent en compte les caractéristiques synchrones (forte réactivité au niveau des comportements, communication par « broadcast », sémantique formelle …). Ces extensions manquent actuellement dans le modèle UML et ses extensions, elles sont prévues dans le modèle UML synchrone que nous avons défini. Nous illustrons sa mise en œuvre par une étude de cas provenant du domaine d'application de l'automobile. Cet exemple, suffisamment riche, nous a permis de cerner les limitations d'UML et nous a donné des idées sur les extensions à apporter.

Les extensions sont de deux types :

- Au niveau des modèles statiques (modèle de classes) il a fallu utiliser la notion de stéréotype pour manipuler des éléments compatibles avec la notion de réactivité des systèmes et de déclenchement d'actions par signaux. Cela a conduit aux *s_capsules.*
- Au niveau des modèles dynamiques les modèles existants ont été étendus pour certains (diagramme de séquences vers les SIB), d'autres ont été remplacés (Statecharts remplacés par SyncCharts). Ceci permet d'avoir une cohérence forte au niveau de leur sémantique et en conséquence, a permis une ouverture vers les outils de preuves. Il est donc possible de faire de la vérification formelle de propriétés intra et inter modèles.

Cela conduit à des réalisations de contrôleurs qui peuvent avoir des parties de code fortement synchrone (composition synchrones des *s*_capsules pour former des *islets)* et très efficace au niveau du code généré. La composition synchrone est réservée aux parties critiques de l'application.
D'autres parties de code (des *capsules),* bien qu'ayant un comportement réactif, peuvent communiquer de manière asynchrone ou asynchrone « borné » avec les islets. Enfin il peut y avoir, comme c'est le cas dans cet exemple, des classes avec appel de méthode classique. C'est le cas en particulier pour ce qui concerne l'utilisation par les parties réatives des fonctions d'interfaçage fournies dans l'exemple.

Les prolongements prévus à ces travaux sont de deux types. A moyen terme, nous développerons une méthodologie liée à UML pour le synchrone. Cela passe par l'étude de nombreux exemples qui fourniront par la pratique, des guides de conception pour les langages synchrones. Pour ce qui concerne les travaux en cours, notre souci est de faire en sorte que nos extensions s'intègrent harmonieusement dans le méta-modèle d'UML, d'autant qu'il existe une forte activité actuellement à l'OMG pour la définition d'UML 2.0.

6 Bibliographie

[ABD98] **C. André, H. Boufaïed, S. Dissoubray**, "*Les SyncCharts : un modèle graphique synchrone pour systèmes réactifs complexes*". RTS 1998, Janvier 1998, Paris, pp175-196.

[ABPRV96] **C. André, F. Boulanger, M.-A. Peraldi, J.-P. Rigault, and G. Vidal-Naquet**. "*Objects and Synchronous Programming"* RAIRO-APII-JESA, Vol. 31, n° 3, 1997.

[And96] **C. André**, "*Representation and Analysis of Reactive Behaviors: A Synchronous Approach*", IEEE-SMC, Computational Engineering in Systems Applications (CESA), 1996, pp 19-29.

[APR01] **C. André, M.-A. Peraldi-Frati, J.-P. Rigault**. "Scenario and Property Checking of Real-Time Systems Using a Synchronous Approach". 4th IEEE International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC 2001), May 2-4, 2001, Magdeburg (D).

[Ber00] **G. Berry**. "The Foundations of Esterel" in Proof, Language, and Interaction. Essays in Honor of Robin Milner. G. Plotkin, C. Stearling, adn M. Tofte, editors. MIT Press, 2000.

[BNLD00] **Y. Le Biannic, E. Nassor, E. Ledinot, and S. Dissoubray**, "*Spécifications Objet UML de Logiciels Temps Réel*". RTS 2000, March 2000, Paris.

[Bou98] **A.Bouali**. "*Xeve: an Esterel Verification Environement", Int'l Conference on Computer-Aided Verification (CAV'98),* June/July 1998, Vancouver, BC Canada. Also available as a technical report INRIA RT-214, 1997.

[Bou93] **F. Boulanger**, "*Intégration de modules synchrones dans la programmation par objets »* Doctoral theis, Supelec University of Paris sud, December 1993.

[Coo00] **S. Cook**, "The UML Family: Profiles, Prefaces, and Packages", Proceedings <<UML>>2000, York, UK, October 2000. LNCS 1939, Springer-Verlag.

[Dou99] **B. Powell Douglass**. "*Doing Hard Time: Developing Real-Time Systems with UML, Objects, Frameworks, and Patterns*". Addison-Wesley, 1999.

[ES] **Esterel Technologies**, *"Esterel Studio",* http://www.esterel-technologies.com
[Hal93] **N. Halbwachs**."*Synchronous Programming of Real Rime Systems »*. Kluwer Academic Publishers, 1993
[Har87] **D. Harel**.*"StateCharts: a Visual Formalism for Complex Systems".* Science of Computer Programming, vol. 8, 1987.
[KN99] **L. Kabous and W. Nebel**. *"Modeling Hard Real-Time Systems with UML: the OOHARTS Approach"*. Proceedings <<UML>>'99, Fort Collins, CO, October 1999. LNCS 1723, Springer-Verlag.
[OMG99] **OMG,** *"OMG Unified Modeling Language Specification. Version 1.3", Object management Group, Inc., Framing-Ham (MA)*, June 1999.
[OMG00] **OMG,** *"Response to the RFP for Schedulability, Performance, and Modeling",* Joint submission by Artisan Tools Inc, I-Logix Inc, Rational Software Corp., Telelogic AB, Time Sys Coorp., Tri-Pacific Software Inc. OMG doc. Number: ad/2000-08-14, August 2000.
|RGG96] **E. Rudolph, P. Graubmann, and J. Grabowski**.*"Tutorial on message sequence charts"* Computer Networks and ISDN Systems, 28(12):1629-1641, December 1996.
[RBP91] **J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, W. Lorensen.** "*Object-Oriented Modeling and Design"* Prentice Hall, Englewood Cliffs, 1991.
[SGW94] **B. Selic, G. Gullekson, and Paul T. Ward**. "*Real-Time Object_Oriented Modelin*g ». John Wiley and Sons, Inc. 1994
[SR98] **B. Selic, J. Rumbaugh**. *"Using UML for Modeling Complex Real-Time Systems"*. http://www.objectime.com, 1998.
[STM] **I-Logix**, *"Statemate MAGNUM",* http://www.ilogix.com
|WTBL00] **Y. Wang, J-P. Talpin, A. Benveniste, P. Le Guernic**. *"a semantics of UML state-machines using synchronous pre-order transition systems"* International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC'2000), IEEE Press, Mars 2000.