DIRECTION DE LA RECHERCHE TECHNOLOGIQUE

CEA/SACLAY
DEPARTEMENT DES TECHNOLOGIES DES SYSTEMES INTELLIGENTS
SERVICE LOGICIELS ET ARCHITECTURES

Service Logiciels et Architectures

Saclay, le xx février 2002

REF. : DTSI/SLA/00-xxx

Transformation ACCORD/UML vers SIGNAL

Par

Yann TANGUY

DRT/LIST/DTSI/SLA/LLSP (CEA)

IDENTIFICATION : Rapport DRT/LIST/DTSI/SLA/00-xxx

TITRE : Transformation ACCORD/UML vers SIGNAL

AUTEUR : Y. TANGUY

MOTS-CLES : Méthodologie UML, objet temps réel, ACCORD/UML, ACCORD, Objecteering, SIGNAL, Langage Synchrone , ACOTRIS, RNTL

UNITE : DRT/LIST/DTSI/SLA

	REDACTEUR	VERIFICATEUR	CHEF DE SERVICE
NOM	**Y. TANGUY**	**S. GERARD**	**D. PLATTER**
DATE			
SIGNATURE			

Résumé

La méthodologie ACCORD/UML a pour but de développer des systèmes embarqués temps réel en utilisant le langage de modélisation UML. Elle est indépendante de tout outil de développement UML (Rose, Rhapsody, Objecteering, …). Dans le cadre du projet RNTL ACOTRIS[1], cette méthodologie sert de base pour une modélisation conforme au standard de modélisation UML, en vue de la connecter avec des langages issus des approches synchrones, en particulier SIGNAL.

Ce document décrit tout d'abord la problématique de traduction d'un modèle UML dans un langage synchrone (en l'occurrence SIGNAL) pour se focaliser dans un deuxième temps sur les choix retenus dans le cadre du projet ACOTRIS.

Enfin, la formalisation de la traduction UML – SIGNAL sera décrite. Celle-ci est basée sur l'utilisation intensive du langage OCL (Object Constraint Language), défini par l'OMG[2] et utilisé ici pour ses capacités à parcourir le méta-modèle de UML. Ce langage est étendu afin de permettre de décrire la transformation de modèle.

Abstract

The goal of the ACCORD/UML methodology is to develop real time embedded systems using UML and real time objects. This methodology is independent from any UML commercial software (like Rose, Rhapsody, Objecteering). The European RNTL labeled ACOTRIS[1] project focuses on merging UML models and implementation models based on synchronous languages, and SIGNAL in particular. ACCORD/UML is the basis methodology used in this project to build fully compliant UML models of real time systems.

This document aims first to describe fundamental translation issues between UML models and synchronous programs. The ACOTRIS viewpoint and every choices relevant to the project are explain in the second chapter.

Last but not least, the translation is depicted in precise details, with a significant attention put on explanations about the language chosen to explicit our translation. This language mainly based on OMG[2]'s OCL, is augmented with some action operator devoted to describe model transformation.

[1] Analyse et Conception à Objets Temps Réel pour Implantation asynchrone/Synchrone http://www.acotris.c-s.fr/

[2] Object Management Group : http://www.omg.org

TABLE DES MATIERES

TABLE DES ILLUSTRATIONS

1 DU SYNCHRONE DANS UML ?

A l'instar des autres familles de langage formel, les langages synchrones, quoique bien connus pour être adaptés à la modélisation de systèmes réactifs, souffrent généralement d'être des langages qui requièrent un haut niveau d'expertise de la part des utilisateurs pour en maîtriser les formalismes sous-jacents. Ce sont généralement des langages dédiés à l'implémentation, et possédant des caractéristiques utiles de vérification de propriétés et de génération de code. Cependant leur utilisation se trouve freinée par leur complexité d'utilisation et les contraintes d'écriture fortes inhérentes à ce type de langage.

UML (Unified Modeling Language [1]) est lui reconnu comme étant un langage de haut niveau offrant un degré d'abstraction suffisant pour permettre à des utilisateurs de décrire des modèles d'application. En tant que langage graphique, UML offre de nombreuses possibilités de représenter un même système, sous différentes vues et à différents niveaux de détail. En contrepartie, il est nécessaire de combiner le langage UML avec une méthodologie dédiée au domaine métier ciblé, d'une part pour guider l'utilisateur dans sa démarche de modélisation et dans l'utilisation des différents diagrammes proposés par UML, d'autre part, afin d'assurer la cohérence et la consistance des différentes vues du système.

Le projet ACOTRIS tente de combiner les avantages de ces deux espaces technologiques via une méthode de modélisation destinée aux systèmes temps-réel, ACCORD/UML. L'idée générale est alors de s'appuyer sur les technologies formelles proposées par SIGNAL et SynDEx pour implanter des applications préalablement modélisées avec ACCORD/UML. De plus, l'idée principale est de mettre en œuvre les technologies formelles liées à ces deux approches sans pour autant demander à l'utilisateur d'en maîtriser tous les tenants et aboutissants. Pour cela, on s'efforcera de définir les transformations de modèles nécessaires à l'automatisation du passage de l'espace technologique UML vers les espaces technologiques de Signal et SynDEx.

La méthodologie ACCORD/UML [2] conçue au CEA-LIST pour le développement de systèmes temps réels embarqués sert de base pour une modélisation conforme au standard de modélisation UML. Elle s'appuie sur la définition d'un profil UML [3] et définit un ensemble d'artefacts et de règles de modélisation permettant la construction des modèles d'une application. Ce dernier est alors soumis à deux transformations (Figure 1~~Figure 1~~) : la première vise à construire un modèle SIGNAL depuis le modèle ACCORD/UML afin de bénéficier de l'outillage de génération de code et de vérification formelle du langage SIGNAL et développés par l'INRIA [4]; la seconde transformation concerne le passage d'un modèle ACCORD/UML vers SynDEx (INRIA) [5]. Elle vise à permettre une implantation temps réel optimisée de ce modèle sur une architecture matérielle donnée.

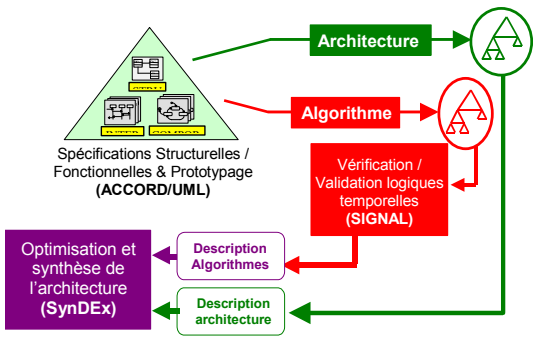

Figure 1 Contours de la plate-forme outil ACOTRIS

Contrairement à l'idée que bon nombre de personnes se font des machines à états de UML, celles-ci ont une sémantique ! Elle est décrite dans la norme [1] (chapitre 2.12.4, page 2-160) et les machines à états-transitions de UML, comme UML dans sa globalité, ne sont donc pas juste un outil de notation graphique, mais bel et bien un langage de modélisation bénéficiant d'une syntaxe concrète et d'une sémantique attachée. La particularité de cette sémantique est qu'elle est paramétrable. En effet, plusieurs points de la sémantique sont dits « points ouverts de variation sémantique ». Ce qui signifie que chaque méthode s'appuyant sur UML doit (ou devrait) préciser les choix faits quant à ces variations possibles de la norme au travers d'un

profil dédié à la méthode. En ce sens, il faut considérer UML comme une base définissant une famille de langages.

Cependant, il faut reconnaître que la plupart des approches « omettent » ce point et laissent implicitement la responsabilité de cette définition aux outils supports sous-jacents pouvant ainsi introduire des problèmes d'ambiguïté sémantique sur les modèles UML.

Loin de nous, cependant, l'idée de prétendre que tout est rose dans UML. En effet, la sémantique de UML présente encore un certain nombre de problèmes de cohérence ou d'ambiguïtés qui font d'ailleurs l'objet d'un grand nombre d'articles [6-9], …) et sont sujets à améliorations à chaque nouvelle mise à jour de la norme. Face à ce problème, trois positions sont alors possibles[3] :

- ne pas utiliser UML ;
- substituer une partie de UML, comme la partie comportementale, par un autre formalisme (ex. : UML-RT/SyncCharts) ;
- définir dans un profil UML spécialisé, encore appelé dialecte UML, un ensemble de règles précisant d'une part les choix faits pour les points ouverts de variation sémantique de UML et d'autre part définissant un ensemble de règles sémantiques supplémentaires, nécessaires pour désambiguïser les points sémantiques « obscurs » de UML.

C'est cette dernière solution qui a été choisie par le CEA-LIST dans le cadre de son travail sur la méthodologie ACCORD/UML. Ce choix permet ainsi de proposer une approche à la fois entièrement conforme au standard UML et pour autant formellement définie. Par formellement, on entend définit de façon à ce que les modèles UML créés en respectant les règles de modélisation définies dans l'approche sont non-ambigus, de plus, les comportements des applications alors produites depuis ces modèles sont déterministes (propriété essentielle dans le contexte des systèmes temps-réel). Ainsi, un effort particulier a été fait dans le cadre de la méthodologie ACCORD/UML pour définir de façon formelle la sémantique des machines à états de UML.

La suite de ce document est ainsi centré sur la description formelle de la transformation d'un modèle ACCORD en un programme SIGNAL.

[3] Il existe une quatrième possibilité : utiliser uniquement la syntaxe concrète de UML et en redéfinir une sémantique. Dans ce cas, on perd tout l'intérêt du caractère « **U**nified » de UML. L'avantage est de disposer d'outil de modélisation du commerce, mais le danger est de tromper les utilisateurs en prétendant être basé sur UML. Car dans ce cas, on ne peut plus parler d'une méthode basée sur le standard.

2 SPECIFICATION DE LA TRANSFORMATION

2.1 Source de la transformation - ACCORD/UML

ACCORD/UML est une méthodologie complètement basée sur UML et visant le développement de systèmes temps-réel par des non-experts du domaine temps réel. Elle repose ainsi sur un ensemble minimal d'artefacts suffisamment abstraits pour être employés par des non-spécialistes du temps-réel tout en permettant une expression des aspects qualitatifs (concurrence, comportement, communication, …) et quantitatifs (contraintes temps-réel, …) des applications à modéliser. On s'est efforcé de construire un ensemble minimal d'extensions car il ne faut pas perdre de vue que l'objectif était de définir une adaptation de UML pour le temps-réel et non pas un nouveau langage. Un profil UML dédié, nommé profil ACCORD/UML, décrit formellement toutes ces extensions pour le temps réel, ainsi que l'ensemble des règles de modélisation définissant les transformations de modèles, les choix pour les points de variation et autres points ambigus de la sémantique UML.

Le paradigme de base de l'approche est le concept d'objets temps-réel. Il est inspiré des langages de programmation concurrente orientés objets tels que [2, 10-12, 13], et étend le concept d'objet actif de UML. L'objectif de cet article n'est pas de décrire en détail les artefacts de modélisation de l'approche. Ces artefacts sont décrits en détail dans la méthodologie ACCORD/UML ([2]).Cependant, afin de comprendre la suite de ce document, on pourra considérer un objet temps-réel comme une entité autonome qui possédant les caractéristiques suivantes:

- communication par échange de messages pouvant posséder une contrainte temps-réel,
- utilisation d'une boîte aux lettres pour stocker les messages reçus,
- et utilisation d'un contrôleur d'état et de contrainte temps-réel liés pour traiter les messages.

En plus des artefacts de base, ACCORD/.UML définit un ensemble de règles de modélisation guidant l'utilisateur durant tout le cycle de développement de l'application, ainsi qu'un ensemble de règles de transformation permettant le passage d'une phase de modélisation à une autre en assurant la continuité du développement. La Figure 2~~Figure 2~~ décrit les principales étapes d'une modélisation selon ACCORD/UML, ainsi que les relations et actions à mettre en œuvre pour passer d'un modèle à l'autre.

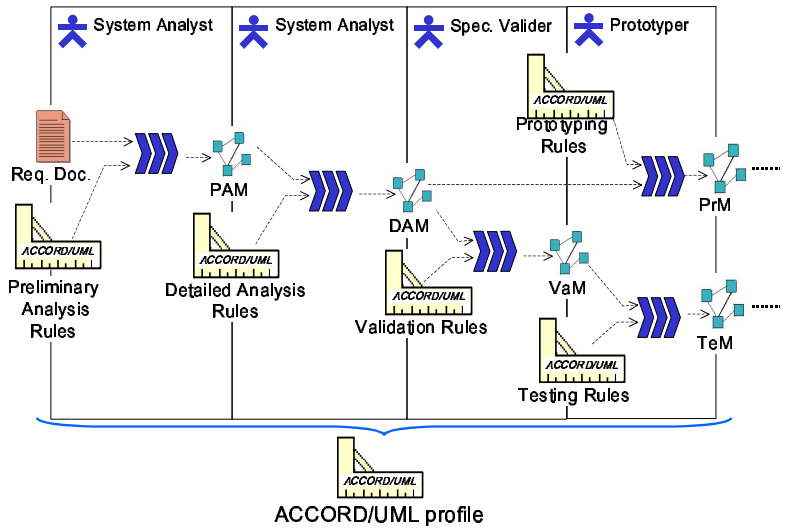

Figure 2 Process de développement ACCORD/UML[4]

Le DAM (Figure 3~~Figure 3~~), ou modèle d'analyse détaillé, est le modèle à analyser pour effectuer la transformation vers SIGNAL. Il contient notamment un modèle structurel des classes de l'application, décrit par des diagrammes de classes, dont nous allons rapidement expliciter l'architecture générale.

ACCORD/UML propose un découpage du modèle structurel en trois parties principales (Figure 3~~Figure 3~~) :

- un paquetage « Provided Interfaces» - il décrit les points d'interaction de l'environnement vers le système ;
- un paquetage « Required Interfaces» - il décrit les points d'interaction du système vers l'environnement ;
- un paquetage « ApplicationCore » - il regroupe l'ensemble des éléments du modèle constituant le cœur du système.

L'exemple de la Figure 3~~Figure 3~~ servira de support illustratif pour le reste de ce document. Il modélise un système doté de deux objets temps réel, Ping et Pong, qui vont tour à tour faire appel l'un à l'autre et décrémenter un compteur pour stopper toute activité une fois ce compteur devenu nul. La valeur de ce compteur est représentée par l'attribut « compteur » de type entier associé à la classe Ping. Ses principales caractéristiques sont :

- Le PingPong démarre lorsque l'on presse le bouton « StartButton »,
- Le compteur s'initialise au démarrage à une valeur fournie par l'interface « StartButton »,
- Le PingPong s'arrête si le compteur devient nul ou si le bouton Stop est pressé,
- Le système affichera la valeur du compteur par une interface « Display ».

[4] Ce cycle de développement utilise le profil SPEM (Software Process Engineering Management)défini par l'OMG

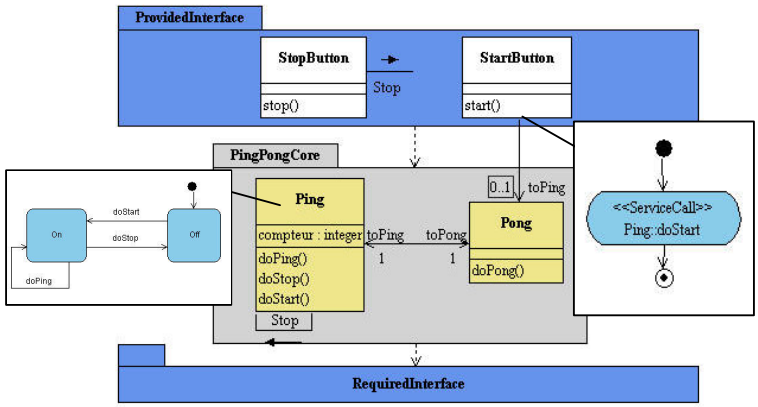

Figure 3 Modèle type d'une application ACCORD/UML simple.

Sous ACCORD/UML, les aspects comportementaux d'une application sont modélisés en attachant aux objets temps-réel d'une application une machine à états-transitions de UML. [14] décrit leur sémantique avec un style opérationnel reposant sur une hypothétique machine d'exécution présentant la configuration suivante (Figure 4~~Figure 4~~) :

- une file d'attente pour stocker les messages entrant sous la forme d'événements et pouvant être de différentes natures (*CallEvent*, *SignalEvent*, …). En ACCORD/UML, les messages reçus possèdent un paramètre particulier sous la forme d'une contrainte temps réel spécifiée par l'émetteur de la communication. Il s'agit en quelque sorte de la qualité de service temps réel requise associée au message ;
- un mécanisme de choix des évènements à dépiler appelé répartiteur d'évènements. Il est à noter qu'il existe ici un point ouvert de variation sémantique. En effet, la politique d'extraction des événements depuis la file d'attente est à définir pour chaque approche. La vue la plus usitée est une politique d'extraction de type FIFO[5], notamment dans les outils qui permettent d'animer les modèles. ACCORD/UML utilise une politique d'extraction de type EDF (« Earliest Deadline First ») comme mécanisme de dépilement des évènements. En effet, les évènements stockés dans la file d'attente d'un objet temps-réel possèdent une contrainte temps réel (échéance, date de début et période). C'est en fonction de cette contrainte temps réel et suivant un algorithme de type EDF que les messages sont choisis dans la file d'attente de l'objet temps-réel ;
- un mécanisme de consommation des évènements (la machine à états-transitions elle-même) qui, en fonction de son état courant, et de l'évènement à consommer, tire éventuellement une transition déclenchant des actions. L'hypothèse d'exécution de tirage est dite « RTC » , pour « Run-To-Completion ». Cela signifie qu'une machine à états-transitions ne traite qu'un seul évènement à la fois. Cependant contrairement à la sémantique synchrone, l'hypothèse de temps nul ne s'applique pas ici. Le tirage d'une transition en UML n'est pas considéré comme une action atomique et peut prendre un certain temps. Pour une modélisation synchrone, il faut décomposer cette action en fonction de ses événements significatifs.

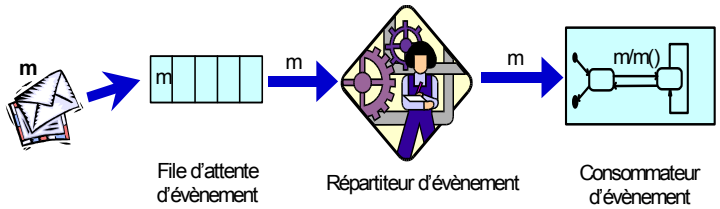

Figure 4 Sémantique des machines à états de UML.

[5] First In First Out

2.2 Cible de la transformation - SIGNAL

Le paradigme synchrone repose sur deux hypothèses principales : les communications se font par des signaux qui sont transmis instantanément et la réaction à un signal se fait sans délai. C'est une vision abstraite du temps (de même que les flottants sont une vision abstraite des nombres réels), représenté comme un domaine discret, ce qui permet des vérifications formelles sur le système modélisé (model-checking, détection de « deadlocks », etc.).

SIGNAL [4] est un langage synchrone déclaratif, avec une sémantique stricte : un processus est un ensemble d'équations sur des flots élémentaires, décrivant à la fois les données et le contrôle. Un flot élémentaire (ou signal) est une suite discrète de valeurs, la succession des éléments d'une telle suite représentant l'écoulement du temps pour ce signal. Lorsque l'on considère l'évolution coordonnée de différents signaux dans un processus, il peut se faire qu'un signal donné soit sans signification (on dira alors qu'il est absent) à un instant où un autre signal est lui présent et possède une valeur significative.

On parle d'*horloges* des signaux pour caractériser les instants de présence relatifs d'un ensemble de signaux. SIGNAL est ainsi un langage multi-horloges dont les opérateurs (qui sont des opérateurs sur des suites : les signaux) permettent d'exprimer des relations sur des signaux ayant une même référence temporelle (une même horloge), ou des références temporelles différentes. En particulier, ces relations peuvent être des propriétés logiques ou temporelles.

L'outil Polychrony (Figure 5~~Figure 5~~) de développement graphique basé sur le langage SIGNAL n'est pas manipulé, ce documents décrit une traduction directe vers SIGNAL, sans passer par un formalisme graphique de type flot de données intermédiaire.

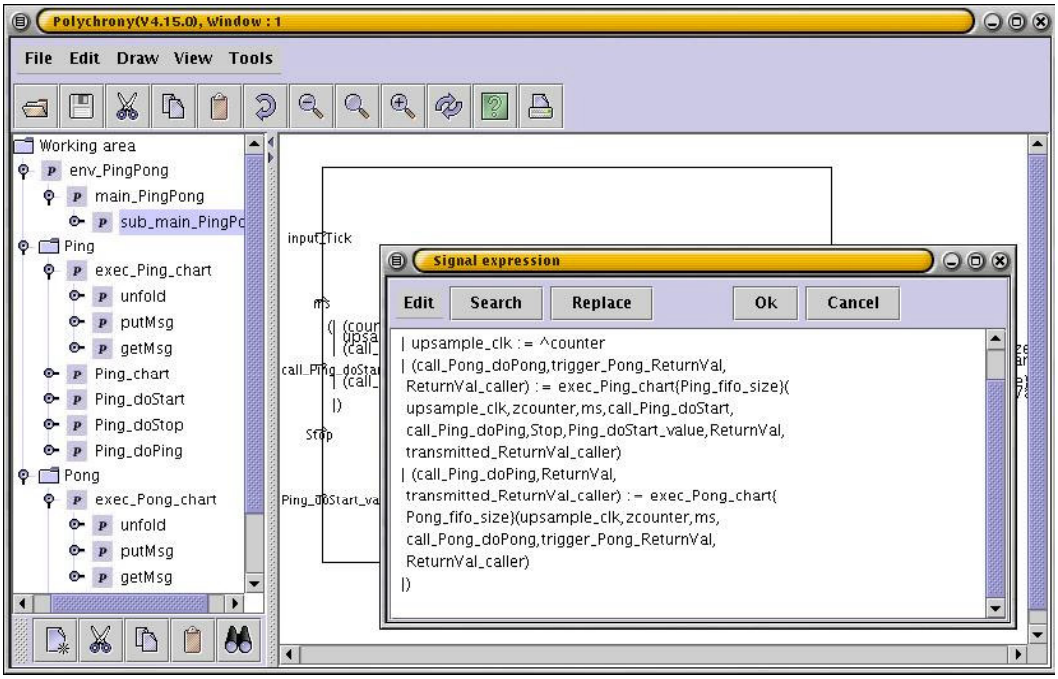

Figure 5 L'outil graphique Polychrony

Dans tous les cas, les fichiers SIGNAL obtenus par la traduction d'un modèle ACCORD/UML pourront être importés dans l'outil Polychrony. Toute information complémentaire concernant SIGNAL et Polychrony peut être trouvée sur le site : http://www.irisa.fr/espresso/Polychrony/ .

2.3 Problématique et choix retenus

Bien qu'ouverte, la sémantique d'exécution des machines d'états de UML ne peut supporter une spécialisation respectant le modèle synchrone : hypothèse de temps non nul d'exécution des transitions ; impossibilité d'avoir des conjonctions d'évènements en déclenchement de transition ; asynchronisme des entités concurrentes d'un modèle UML comme les objets actifs, les états concurrents et les activités ; ….

Contrairement à [15], notre objectif dans la définition de la transformation de modèle vers SIGNAL n'est pas de définir une sémantique synchrone à UML mais de définir un patron de conception en SIGNAL permettant de supporter le modèle d'exécution asynchrone tel que défini dans ACCORD/UML. Cette approche nous nous place dans la catégories des systèmes globalement asynchrones / localement synchrones, pour lesquels de nombreux travaux sont en cours afin de formaliser ce couplage et de donner une méthodologie adaptée, et de mieux cerner les possibilités de vérification sur les système asynchrones / synchrones.

En particulier, il s'agit de traduire l'asynchronisme lié aux files d'attentes dans un processus SIGNAL, ce qui est possible dans la mesure où l'on se limite à considérer des files d'attente de tailles bornées. Modulo cette restriction, l'abstraction synchrone du modèle SIGNAL peut s'appliquer à un modèle ACCORD/UML. Le modèle d'exécution de UML est traduit vers signal afin de conserver le sens du modèle UML initial. En ce sens, l'exercice de traduction d'un modèle UML vers SIGNAL peut être vue comme la formalisation de la sémantique de UML à l'aide du langage SIGNAL.

i. Une approche MDA

D'autre part, la définition de mécanismes pour passer d'un modèle UML à un programme SIGNAL entre dans le cadre des travaux du CEA autours de ACCORD/UML et du MDA. MDA signifie « Model Driven Architecture » [16], i.e. l'architecture guidée par les modèles. MDA est fortement lié à deux autres mots clés : le « PIM » et le « PSM » (cf. Figure 6~~Figure 6~~). Dans ces deux acronymes, la lettre « P » signifie Plate-forme qui fait référence « aux éléments techniques et structurels qui sont sans rapports avec les fonctionnalités de base d'un élément logiciel » . Le PIM, pour « Platform Independent Model », est un modèle dans lequel sont définis et manipulés seulement des éléments qui sont totalement indépendants de toute implémentation logicielle et technique. L'avantage évident d'un tel modèle, usuellement qualifié de modèle métier, est que son indépendance vis-à-vis des technologies d'implémentation facilite son portage vers différents environnements opérationnels basés sur des technologies d'implémentation pouvant être tout à fait différentes. Le modèle résultant d'un tel portage est alors appelé le PSM, c'est à dire « Platform Specific Model ». A ce stade, des choix technologiques d'implémentation ont été faits et le PIM doit être implémenté sur ou selon un PSM. On peut par exemple imaginer un PIM qui serait transformé en un modèle PSM1 basé sur une implémentation C++ sous VxWorks, et une autre configuration PSM2 qui utiliserait le langage Java avec Windows NT.

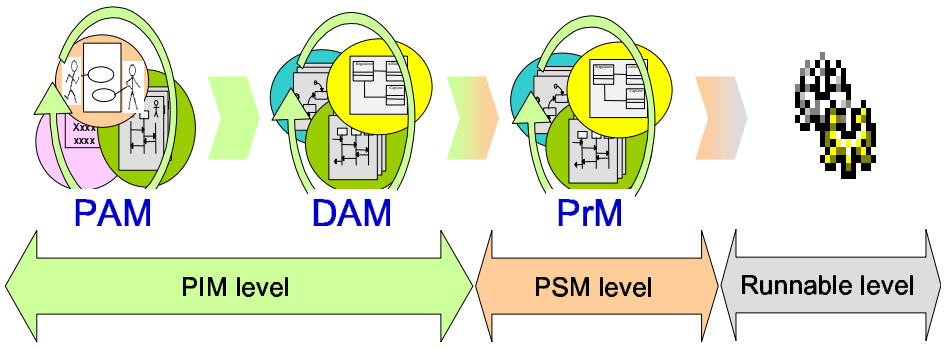

Figure 6 Le PIM et le PSM dans ACCORD/UML

Le premier résultat attendu de l'application du paradigme MDA pour les ingénieurs est de faciliter le portage d'applications existantes modélisées sous la forme d'un PIM vers d'éventuelles nouvelles technologies plus performantes, plus sûres, … (langages d'implémentation, middleware, etc.), et tout cela à moindres efforts, risques et coûts bien sûr.

L'utilisation des modèles pour construire des logiciels n'est bien entendu pas nouvelle en soit. Ce qui l'est plus est cette idée d'indépendance/dépendance des modèles vis-à-vis des technologies d'implémentation et des technologies sous-jacentes et primordiales de manipulation des modèles (transformation, gestion de configuration, traçabilité des modèles, …). L'importance majeure de cette initiative ne réside donc pas tant dans son caractère novateur que fédérateur. En effet, elle a permis de faire émerger/reconnaître une activité du génie logiciel qu'est l'ingénierie des modèles et stimulé ainsi toutes sortes d'initiatives sur le sujet : workshops UML, OOIS, ECOOP, Groupe OFTA sur l'ingénierie des modèles, projets de recherche nationaux et européens, …

Actuellement, il semble que du point de vue méthodologique deux grandes orientations se dessinent : les approches basées MOF (Meta-Object Facility) et les approches basées profils UML.

ii. MDA par méta-modélisation MOF

La méta modélisation est utilisée pour construire un méta modèle spécifique à un cadre d'application ou un domaine particulier. L'OMG a spécifié un méta meta modèle, MOF [14], afin d'avoir un langage commun de description de méta modèle. Le MOF permet de décrire un nouveau méta modèle par extension (ajout de méta classes, méta relations…) d'un méta modèle existant, propriété qui peut permettre d'obtenir une certaine réutilisabilité des méta modèles MOF. La méta modélisation apparaît comme une solution spécifique à un problème précis, une réponse parfaitement adaptée, mais au détriment de la flexibilité et de la simplicité d'utilisation.

En effet, le principal obstacle à la mise en œuvre d'un méta modèle MOF réside dans le fait qu'il n'existe pas (peu ?) d'outils capables à partir d'un tel méta modèle de fournir un environnement graphique de modélisation associé. De plus, la création d'un nouveau méta modèle stable, est une étape nécessaire pour définir les transformations

iii. MDA par « profilage » UML

La définition d'un profil UML consiste en une spécialisation du méta modèle UML pour l'adapter à des besoins ou un domaine d'utilisation particulier. Cette notion repose sur les possibilités de spécialisation fournies par UML, comme les stéréotypes, les contraintes ou les « tagged values ». Un profil UML ne modifie pas son méta modèle, qui reste conforme à la norme définie par l'OMG.

Dans la mesure où les profils reposent sur le Meta modèle UML, les modèles écrits selon tel ou tel profil restent des modèles UML, ce qui permet des transformations plus simples et plus rapides à mettre en œuvre. De plus, l'écriture d'un profil adapté à un domaine ne nécessite pas l'écriture d'un méta modèle complet, il suffit, partant du méta modèle UML, de compléter celui-ci par les notations ou concepts liés au domaine concerné. Dans le cadre de MDA, le PIM est souvent un modèle UML utilisé avec un profil « métier », c'est-à-dire un profil qui fournit les concepts nécessaires à un domaine métier particulier, tandis que les PSM, directement liés à l'implémentation, seront par exemple des modèles UML utilisant des profils de génération de code (C++, Java, …). Il arrive couramment que l'on ait besoin de transformer le PIM en PSM, d'où l'intérêt d'avoir des modèles reposant sur des méta modèles proches.

Enfin, les utilisateurs commencent à être réellement familiarisés avec UML et les AGL qui reposent sur ce méta modèle (Rational Rose, Objecteering, …). L'outil Objecteering de Softeam permet l'écriture de profils UML [17] et l'utilisation de ceux-ci dans Objecteering. Il est probable que la position de l'OMG en ce qui concerne les profils amène les autres éditeurs d'outils à intégrer cette fonctionnalité dans leur AGL.

iv. D'un PIM ACCORD/UML vers un PSM SIGNAL

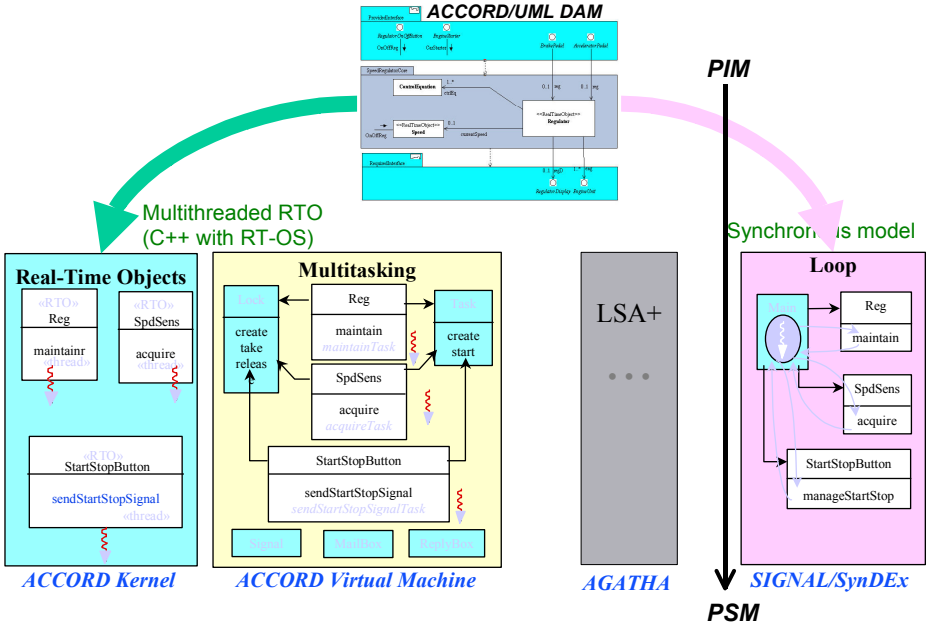

Figure 7 Différents PSM possibles pour le DAM

Dans la méthodologie ACCORD/UML, les étapes de modélisation PAM et DAM correspondent à une modélisation de niveau PIM, soit indépendante de la plate forme visée, ainsi que du langage d'implémentation qui sera utilisé. Le passage du DAM au PrM (modèle de prototype) est une transformation PIM vers PSM, dans la mesure ou on doit choisir la plate forme cible (la machine virtuelle ACCORD en général) et le langage d'implémentation (le C++ quand on vise la plate forme ACCORD). Dans le cadre du projet ACOTRIS, le langage ciblé est le langage SIGNAL, il faut définir une transformation de modèle permettant de passer d'un modèle d'analyse UML (DAM) à un modèle (programme) SIGNAL. Il s'agit d'une transformation exomorphe, c'est à dire que le formalisme de départ et celui d'arrivée sont différents, en d'autre termes, les modèles de départ et d'arrivée sont chacun une instanciation d'un meta modèle différents. Quel que soit le langage utilisé pour décrire une telle transformation, il doit posséder les propriétés suivantes :

- Capacité à parcourir le méta modèle de départ
- Capacité à parcourir le méta modèle d'arrivée
- Moyen de créer, modifier des éléments du meta modèle d'arrivée

Dans le cas d'un modèle UML, le problème ne se pose pas, le meta modèle existe (Figure 8~~Figure 8~~), il s'agit d'UML lui même. Dans le cas du langage SIGNAL (qui est par ailleurs représentatif du cas général d'une transformation exomorphe), il n'existe pas de meta modèle, il faut avant tout le créer si on veut être capable de décrire correctement la transformation. Attention, il ne s'agit pas ici de décrire un meta modèle doté d'une sémantique synchrone, nous cherchons seulement à décrire un meta modèle de la syntaxe du langage SIGNAL. La solution idéale est dans ce cas de se baser sur la grammaire même du langage pour définir un meta modèle « syntaxique » complet. Notre approche est ici plus pragmatique, le meta modèle SIGNAL que nous utilisons ne décrit pas complètement, la syntaxe du langage mais seulement les artefacts minimaux nous permettant de décrire la transformation.

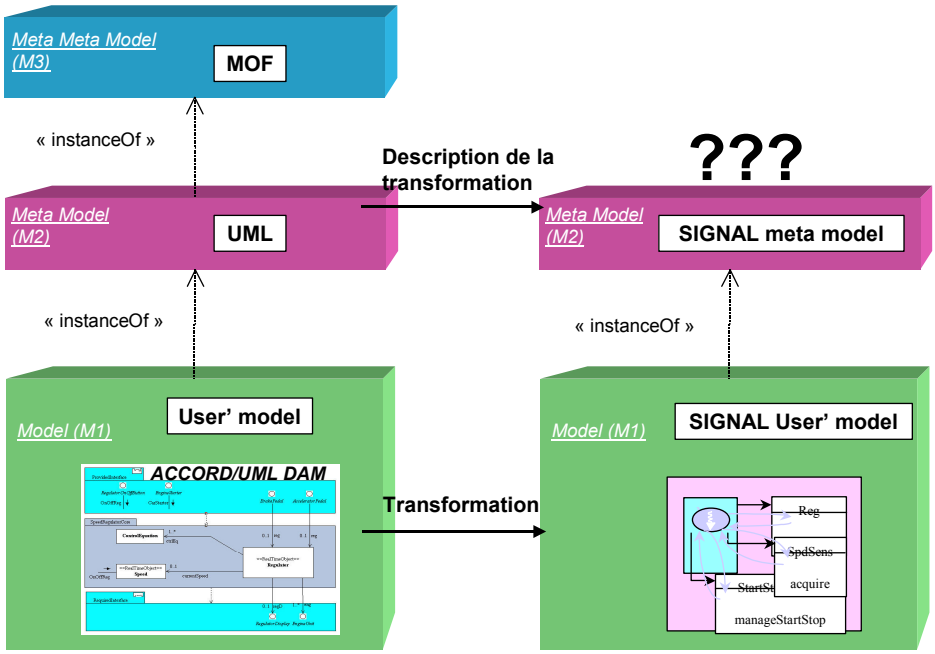

Figure 8 Organisation de la transformation de modèles

Afin de réduire la complexité de la transformation, celle ci est basée sur un modèle ACCORD/UML DAM simplifié. De la même façon que ACCORD/UML utilise une restriction des machines à états de UML pour résoudre les problèmes d'ambiguïté sémantique, la transformation s'appliquera à des modèles DAM simplifiés pour se débarrasser de certaines difficultés de traduction. Ces règles seront décrites dans le chapitre suivant.

Tel qu'il est définit dans ACCORD/UML un message reçu est stocké avec une contrainte temps réelle propre (échéance, périodicité, latence,…) puis extrait selon un algorithme de type EDF[6], c'est à dire que la requête la plus urgente (échéance proche) est placée prioritairement par rapport aux autres, indépendamment de son ordre d'arrivée dans la boite de réception.

Dans un premier temps pour faciliter le passage vers SIGNAL, nous utiliserons en phase de modélisation UML, non pas des boîtes aux lettres comme mécanisme de gestion des messages d'un objet tel que défini dans ACCORD, mais une file d'attente de type FIFO. D'une part l'écriture de ce mécanisme est plus simple et peut se faire entièrement en SIGNAL, sans modifier fondamentalement le schéma global de la traduction. D'autre part, il sera possible après validation de la traduction, de modifier le modèle d'exécution, en changeant l'implémentation des processus SIGNAL dédiés à l'écriture et à la lecture dans la boite de réception, éventuellement en en donnant une implémentation externe dans un autre langage (C, C++, Java…). De plus afin d'obtenir une transformation générique, les mécanismes de traduction mis en place seront tels que le passage à un autre mode de gestion puisse se faire facilement. Dans ce contexte on étudiera dans un second temps la prise en compte de modèles plus complexes comme la boîte aux lettres.

[6] Earliest Deadline First

3 REGLES DE MODELISATION

Les règles de modélisation indiquées dans ce chapitre sont de deux sortes. Certaines proviennent directement des choix techniques faits dans cette version de la traduction, comme le choix de la file d'attente FIFO pour traiter les messages, ou les contraintes sur les types de données manipulés. Les autres peuvent être vues comme des règles de bonne modélisation afin de vérifier qu'un modèle est traduisible vers SIGNAL.

Ces règles reposent sur la méthodologie ACCORD/UML, mais en restreignent certains aspects, ou imposent certaines règles au niveau de la phase de modélisation de l'analyse détaillée du système. Elles définissent en ce sens une annexe de la méthodologie ACCORD/UML liée à l'objectif d'implémentation synchrone.

3.1 Architecture générale du modèle UML

Le modèle auquel on fait ici référence est le Modèle d'Analyse Détaillé (en court DAM pour « Detailed Analysis Model ») tel que décrit dans la méthodologie ACCORD/UML (L'objet de ce document n'étant pas de re-décrire la méthode ACCORD/UML elle-même, pour plus de détails sur l'approche, le lecteur pourra se référer à [18]).

Nous ne traiterons dans cette section que les contours généraux de l'approche avant de donner les règles de bonne modélisation du modèle UML vu dans l'optique de la traduction vers SIGNAL. Les points généraux d'un modèle DAM ne seront pas détaillés, toutes les informations nécessaires étant dans le document décrivant la méthodologie ACCORD/UML. Nous n'aborderons ici que les différences que l'utilisateur doit connaître lors de sa modélisation.

La Figure 9~~Figure 9~~ présente l'architecture générale, au niveau de l'analyse détaillée. Dans la suite du document, nous appellerons ce modèle DAM simplifié afin de le différencier des modèles DAM décrits par la méthodologie ACCORD/UML.

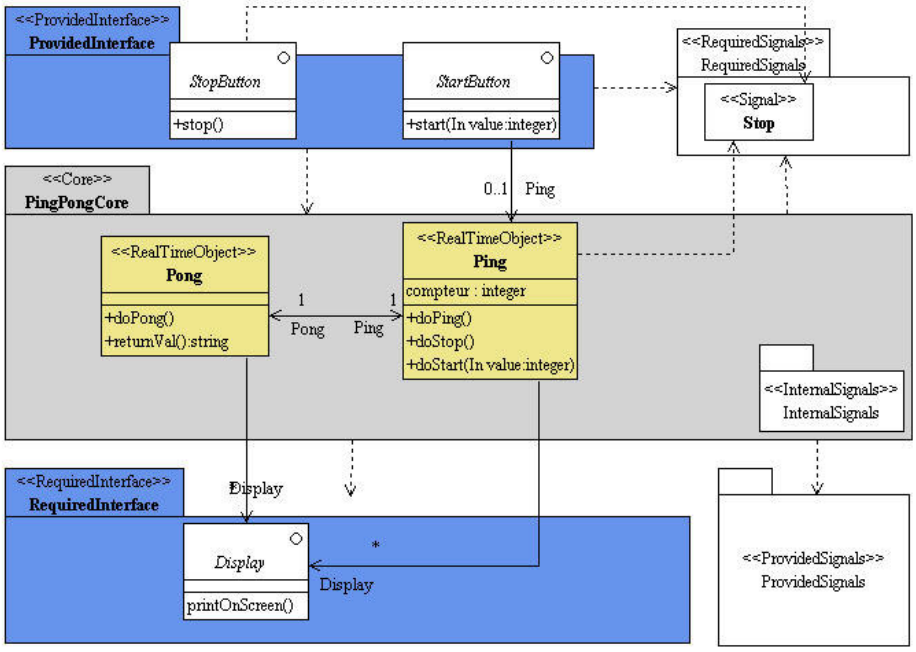

Figure 9 Architecture standard d'un modèle ACCORD UML – SIGNAL

Le DAM simplifié est un modèle principalement composé de trois couches :

- Un paquetage stéréotypé « ProvidedInterface » - contient les interfaces fournies par le système à son environnement.

- Un paquetage stéréotypé « Core » - contient toutes les classes qui composent le système à proprement parler, et en particulier les classes temps réel.
- Un paquetage stéréotypé « RequiredInterface » - contient les interfaces que l'environnement est sensé fournir à l'application pour permettre son fonctionnement normal.

Ces paquetages ont le même rôle que dans le DAM, ils sont complétés de paquetages regroupant les différents types de signaux[7] de l'application :

- Un paquetage stéréotypé « ProvidedSignals » - contient la description des signaux fournis par le système à son environnement.
- Un paquetage stéréotypé « InternalSignals » - contient les signaux purement internes au système.
- Un paquetage stéréotypé « RequiredSignals » - contient la description des signaux que le système attend de l'environnement pour fonctionner.

Le rôle de ces paquetages étant défini par typage, ils ne peuvent être cumulés : un paquetage ne peut pas être à la fois containeur des interfaces requises et des interfaces fournies.

Dans cette version de la traduction, nous ne distinguons pas objet et classe, (implicitement, une classe possède une instance unique). Toutes les communications et références aux objets se feront en spécifiant directement le nom de la classe ciblée. Il s'agit d'une restriction qui nous permet de réduire les diagrammes manipulés dans la transformation, en l'occurrence, on s'abstient ainsi d'utiliser les diagrammes d'objets. Cette restriction est forte du point de vue de ma modélisation objet, toutefois, il reste aisé de passer du diagramme de classe DAM vers celui du DAM simplifié par copie et renommage des classes possédant plusieurs instances.

Les signaux seront spécifiés par des classes stéréotypées « Signal ». Toute classe susceptible d'envoyer un signal donné se doit d'être liée à la classe qui représente ce signal, de même que toute classe temps réel voulant recevoir un signal. Le type de relation utilisé est dans ce cas un « lien d'utilisation » UML.

Pour simplifier nous pouvons dire que toutes les classes qui communiquent entre elles par appels d'opération, doivent être reliées entre elles au moyen d'une association (comme sur la Figure 10~~Figure 10~~), tandis que les classes qui communiquent par signaux doivent simplement déclarer un lien d'utilisation vers les signaux concernés (comme sur la Figure 11~~Figure 11~~).

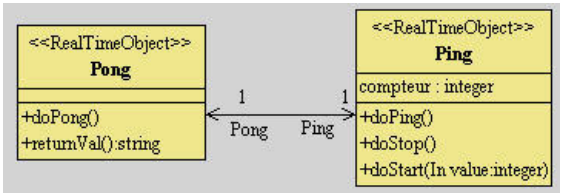

Figure 10 Association entre classes

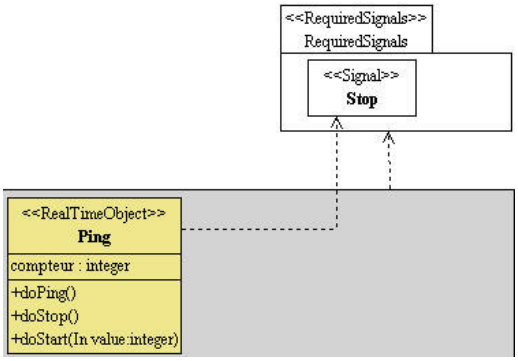

Figure 11 Lien d'utilisation entre classes

3.2 Appels de service entre objets

3.2.1 Description des services

Actuellement, le comportement des services (et plus généralement de toutes les opérations) est décrit dans le modèle dans des notes de texte contenant des portions de code C++. ACCORD/UML fait ce choix, d'une part parce qu'il n'existe pas encore de langage approprié dédié parfaitement normalisé et supporté par les outils pour représenter l'algorithmique des opérations (l'Action Semantic Language qui vise cet objectif est pour une bonne part encore en

[7] Un signal a ici le sens qui lui est donné par la méthodologie ACCORD/UML

travaux à l'OMG),d'autre part, le langage cible des prototypes ACCORD/UML étant le C++, il est logique de décrire l'algorithmique des opérations directement dans ce langage.

De part la structure même du langage SIGNAL et les contraintes d'écriture fortes qu'il impose, il apparaît comme très difficile de parcourir un code écrit en C++ pour en faire une traduction vers signal.

La description précise des opérations se fera à l'aide de notes contenant du code SIGNAL (Figure 12~~Figure 12~~), et sera directement intégrée au code SIGNAL généré à partir du modèle.

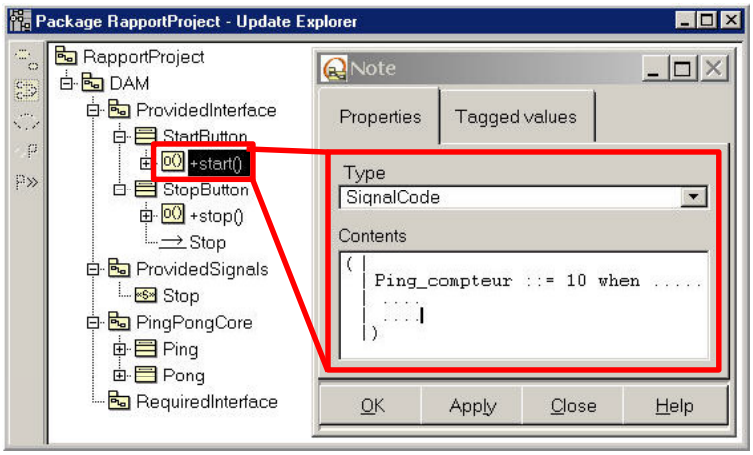

Figure 12 Note de type SignalCode

Cependant, si les appels aux opérations des classes du système peuvent être appréhendés comme des appels à des processus SIGNAL, il n'en va pas de même dès que l'on fait appel aux services des classes temps réel. En effet, dans le modèle d'exécution de ACCORD/UML, l'appel à un service d'une classe temps réel résulte en l'écriture d'un message dans la boîte de réception de l'objet ciblé. Dans le même esprit, un signal doit être inscrit comme message dans les boîtes de réception de tous les objets qui se sont déclarés sensibles au signal.

Ceci implique que dans le DSM les appels de services et les créations de signaux doivent être explicités, pour traitement lors de la traduction.

3.2.2 Appels de services et création de signaux

Les appels de service provenant d'une opération, sont décrits par un diagramme d'activité associé à cette opération. Il prennent la forme d'un « action state » ayant pour nom : Objet ::Nom_service(param), où Objet est le nom de l'objet temps réel ciblé, Nom_service le nom du service appelé et param la liste de ses paramètres. Cette action state doit être stéréotypé « ServiceCall » (Figure 13~~Figure 13~~) afin de le différencier simplement des autres action state éventuels. Il n'est pas impératif de décrire l'algorithme complet des opérations dans les diagrammes d'activité, seuls les appels de service sont obligatoires. L'appel à une opération d'une classe non temps réel, suit la même syntaxe, mais n'est pas stéréotypé « ServiceCall », et n'est pas traité lors de la traduction, il n'a à ce niveau, qu'une valeur descriptive.

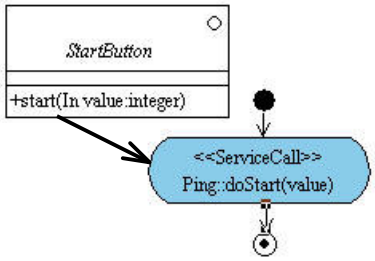

Figure 13 Un action state stéréotypé "serviceCall"

Dans l'exemple Figure 13~~Figure 13~~, le diagramme d'activité nous signale que dans l'opération *start()* de la classe *StartButton*, un appel est fait au service *doPing()* de la classe temps réel *Ping*, un paramètre *value* est passé à *Ping* lors de l'appel.

Dans le cas d'un appel de service avec valeur de retour, l'implémentation du modèle ACCORD/UML déclare une Rbox, ou boîte de réception du type retourné par le service appelé. Dès que le service est exécuté, la valeur retournée devient disponible dans la Rbox déclarée par l'appelant. Ceci permet de différencier des appels de service « bloquants » dans lesquels l'exécution de l'appelant est suspendue en attente de réponse. L'exécution de l'appelant peut continuer, et récupérer la valeur de retour quand elle est prête (test de disponibilité). Au niveau de l'appel du service, ceci est représenté par l'affectation typée :

Type variable_de_reception = Service_avec_retour(paramètres) (Figure 14~~Figure 14~~)

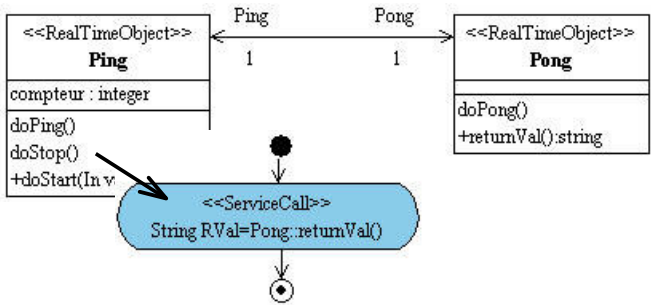

Figure 14 Appel de service avec valeur de retour

Dans l'exemple ci-dessus *Ping* appelle le service *returnVal* de *Pong* et stocke la valeur de retour dans une variable locale *Rval*, dès que celle-ci est disponible.

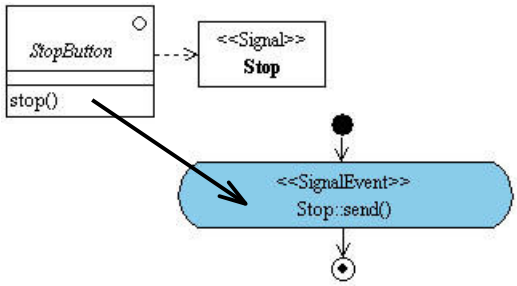

Figure 15 Envoi d'un signal Stop

De même, les envois de signaux doivent apparaître en tant qu'activity state stéréotypé « signalEvent » dans les diagrammes d'activité associés aux opérations (Figure 15~~Figure 15~~).

La syntaxe est : Nom_Signal ::send(paramètres)

L'envoi du signal est possible s'il existe un lien depuis la classe qui poste le signal vers la classe qui spécifie le signal. Il est possible de passer un paramètre associé au signal, à condition qu'un attribut du même type représente ce paramètre dans la classe qui spécifie le signal. Dans l'exemple Figure 15~~Figure 15~~, le diagramme d'activité nous signale que dans l'opération *stop()* de la classe *StopButton*, le signal *Stop* est envoyé dans le système.

3.2.3 Valeurs marquées RTF et types de données

L'utilisateur peut associer une contrainte temporelle à ces action state particuliers, celle-ci prend la forme d'une valeur marquée de type RTF[8]. Dans ACCORD/UML, tout appel de service à un objet temps réel doit avoir une contrainte RTF associée, quand celle-ci n'est pas définie, elle prend la valeur de celle qui est associée au service appelant. Cette version de la traduction vers signal, utilisant un mécanisme de type FIFO pour gérer les messages, les contraintes RTF seront ignorées, et n'auront de fait qu'une valeur informative.

[8] Real Time Feature

Autre point important, par défaut, les types autorisés dans le modèle DSM seront ceux définis dans le langage signal. L'utilisateur peut toutefois utiliser de nouveaux types à condition de pouvoir les définir en signal. D'autre part, tout type utilisé comme paramètre dans un appel de service doit avoir les fonctions de conversion *type2string* et *string2type* d'un type donné sous forme de chaîne de caractères et vice versa, afin de permettre l'écriture du paramètre sous la forme d'une chaîne dans la FIFO de réception des messages.

3.3 Comportement des objets temps réel

Dans cette version de la traduction ACCORD/UML – SIGNAL, la file d'attente des machines à états est de type FIFO. Il s'agit d'une simplification liée aux difficultés d'écriture de la boîte de réception des messages de ACCORD/UML en SIGNAL. En utilisant une file d'attente FIFO, on peut la décrire complètement à l'aide de processus SIGNAL (sans utiliser de code C externe). Il en résulte que les contraintes temporelles de type RTF éventuellement spécifiées ne seront pas prises en compte dans cette traduction. Elles gardent toutefois, pour le développeur, une valeur informative, liée aux spécifications du système à modéliser.

Ce choix reste conforme à la norme UML qui impose l'utilisation d'une boîte de réception des messages, sans décrire sa mise en œuvre (point ouvert de variation sémantique). Le concepteur doit garder en mémoire le modèle d'exécution sous-jacent (lié à la sémantique des machines à états qu'il manipule) durant la modélisation. Le traitement des messages par FIFO, l'hypothèse RTC ou encore le traitement des évènements quand ils ne sont pas éligibles, sont autant de contraintes qui conditionnent et modifient fondamentalement l'interprétation que l'on fait d'une machine à état-transitions.

On considère que tout message extrait de la file d'attente, s'il n'est pas consommable par la machine à état, c'est-à-dire s'il n'existe aucune transition autorisée pour l'évènement depuis l'état courant, est définitivement perdu. Cette version du traducteur ne prend en compte que des machines à états non hiérarchiques, il s'agit d'une contrainte de modélisation forte mais non réductrice : en effet, il est toujours possible de transformer une machine à états hiérarchique en machine à états non hiérarchique équivalente (Figure 16~~Figure 16~~).

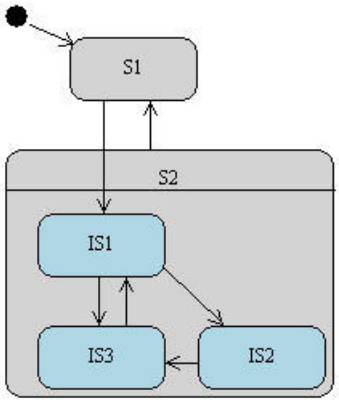

Figure 16 Une machine à état hiérarchique

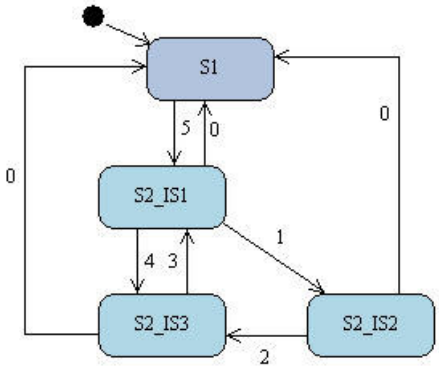

Figure 17 Simplification non hiérarchique de la machine à états de la figure de gauche

A ces réductions près, l'utilisation qui est ici faite des machines à états-transitions de UML ne diffère pas de celle décrite dans la méthodologie pour le DAM d'un modèle ACCORD/UML standard. A savoir que cela concerne les classes temps réel, qui ont toutes une machine à états-transitions associée qui décrit leur comportement. ACCORD/UML propose de donner deux vues d'une machine à états :

- Une vue protocole – qui décrit « ce que l'objet peut faire », en bref tous les appels de services autorisés.(cf. Figure 18~~Figure 18~~)
- Une vue triggering – qui décrit « ce que l'objet doit faire », c'est à dire la réaction aux signaux et évènements.(cf. Figure 19~~Figure 19~~)

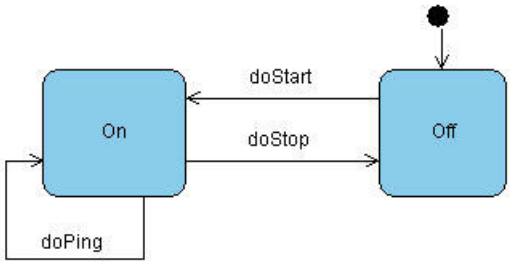

Figure 18 Vue protocole de la machine à état de la classe *Ping*

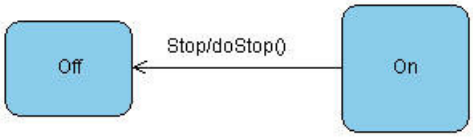

Figure 19 Vue triggering de la machine à états de la classe *Ping*

4 FORMALISATION DE LA TRANSFORMATION

Cette version de la traduction écarte :

- les machines à états hiérarchiques
- On ne distingue pas class et instance → attention aux noms !!!
- les contraintes temporelles ACCORD/UML (utilisation d'une FIFO)

Ces contraintes sont temporaires et ne constituent pas de problèmes majeurs de traduction, elles pourront être résolues dans les prochaines versions du traducteur.

4.1 Architecture générale du modèle SIGNAL

Un programme SIGNAL est un processus, qui possède des entrées et des sorties, et éventuellement des sous-processus. Les processus du langage SIGNAL, vont représenter les opérations associées aux classes du DSM. SIGNAL propose de regrouper un ensemble de processus dans des modules. L'utilisation de modules permet le regroupement de processus, mais ne définit pas de sémantique associée, il s'agit d'un simple ensemble. Les modules nous serviront dès lors à grouper ensemble tous les processus représentant les opérations d'une classe. Une classe du modèle UML sera généralement traduite par un module.

Comme décrit Figure 20~~Figure 20~~, le modèle SIGNAL est constitué d'un processus principal dont les entrées sont les évènements externes que le système peut recevoir. Ceux-ci correspondent à l'ensemble des services proposés par les interfaces contenues dans le paquetage stéréotypé « ProvidedInterface ». Ce paquetage gère les évènements externes au système, et fait appel en parallèle aux processus qui gèrent l'exécution de chaque objet temps réel.

```
process env_PingPong =
  { integer Ping_fifo_size, Pong_fifo_size; }
  ( ? boolean Bms;
      boolean BStart;
      boolean BStop;
    ! …;)
  (| Bms ^= BStart ^= BStop ^= Tick
   | …
   | main_PingPong(Tick, ms, call_Ping_doStart, Stop, Ping_doStart_value)
   |)
  where
  event Tick, ms, …

process main_PingPong =
  ( ? …
    ! …)
  (| sub_main_PingPong(Tick, ms, call_Ping_doStart, Stop, Ping_doStart_value)
   | …
   |)
  where
    statevar integer Ping_compteur init 10;
    constant integer nb_inserted = 4;

    type code_sender = enum (dummy, Ping_doStop);

    process sub_main_PingPong =
      ( ? …
        ! …
      )
      (| (counter, zcounter) := upsample(nb_inserted when input_Tick, input_Tick)
       | upsample_clk := ^counter
       | (call_Pong_doPong, … ) := exec_Ping_chart{Ping_fifo_size} ( …, call_Ping_doStart, … , Stop, …)
       | (call_Ping_doPing, … ) := exec_Pong_chart{Pong_fifo_size} ( …, call_Pong_doPong, …)
       |)
      where
        integer counter, zcounter;
        …
        event call_Pong_doPong, …

        use Ping;
        use Pong;

      end;
  end;
end;
```

Figure 20 Processus principal de l'exemple PingPong

Les processus exec_xxx_chart sont chargés :

- d'écrire les messages reçus dans leur file d'attente propre,
- de dépiler les messages en fin de pas RTC (fin de l'exécution du service précédent),
- de décider si le message est acceptable ou non (traduction signal de la machine à états),
- d'exécuter les services demandés quand l'état courant le permet.

Les sous-processus du processus principal ont globalement un rôle d'initialisation et de déclaration de variables.

L'entrée ms représente l'horloge du système, les autres entrées sont les interfaces fournies par le système.

Le processus *exec_xxx_chart* (cf. Figure 21~~Figure 21~~) gère globalement la lecture et l'écriture des messages dans la boîte de réception. Les processus concernés sont :

- *unfold*
- *putMsg*
- *getMsg*

- *FIFO*

```
module Ping =

type Ping_code_msg = enum (call_Ping_doStart, call_Ping_doStop, call_Ping_doPing, Stop);
type Ping_message = struct (Ping_code_msg id; integer param; integer deadline;);


process exec_Ping_chart =
  { integer Ping_fifo_size; }
  ( ?
    !
  )
  (| (shifted_Stop, shifted_Ping_doStart, … ) := unfold( … , upsample_counter, Stop, call_Ping_doStart, … )
   | …
   | msg_in := putMsg(shifted_Stop, shifted_Ping_doStart, … )
   | (msg_out, alarm_empty, … ) := FIFO{Ping_message, Ping_fifo_size} (upsample_clk, msg_in, clk_getMsg)
   | first_msg := (not (^first_msg)) $ init true
   | first_msg ^= when OK_put
   | (provided_Stop, provided_Ping_doStart, … ) := getMsg(msg_out, OK_get)
   | no_input_event := when (not OK_get)
   | (end_step, call_Pong_doPong, … ) := Ping_chart( … , provided_Stop, provided_Ping_doStart, …)
   | ReadyToExecute := (when first_msg) ^+ end_step ^+ no_input_event
   | clk_getMsg := shift_event(ReadyToExecute, upsample_clk)
   |)
  where
    boolean OK_put, OK_get, first_msg;
    event alarm_empty, alarm_full;
    …

    process unfold =

    process putMsg =

    process getMsg =

end;
…
```

Figure 21 Module de la classe temps réel *Ping* (partie 1)

Le processus *unfold*

Le modèle synchrone, basé sur un temps virtuel discret, prend en compte la simultanéité possible des événements dans ce temps logique : des messages peuvent arriver simultanément en entrée d'une file d'attente donnée. Afin de pouvoir traiter séquentiellement leur ajout dans la file et ainsi n'en perdre aucun, une solution offerte par SIGNAL consiste à utiliser un *sur-échantillonnage*. On crée une horloge interne (d'entrée des événements dans la file) plus rapide que l'horloge de ces événements.

Ce processus prend en compte la simultanéité éventuelle des évènements dans un programme SIGNAL (et plus généralement avec les langages synchrones). Pour ne pas perdre d'évènements, lors de l'écriture dans la boîte de réception, le processus *unfold* fait un décalage temporel des évènements en entrée du processus afin de pouvoir les écrire séquentiellement dans la boîte de réception. Il utilise un paramètre correspondant au nombre maximal d'entrées traitées simultanément par un même objet temps réel. Cela revient à sur-échantillonner l'horloge principale du système. La Figure 22~~Figure 22~~ illustre ce processus dans le cas du module *Ping*, les évènements reçus par le processus sont décalés de manière à les rendre séquentiels.

```
process unfold =
    ( ? event upsample_clk;
        integer upsample_counter;
        event Stop, call_Ping_doStart, call_Ping_doPing, call_Ping_doStop;
      ! event shifted_Stop, shifted_Ping_doStart, shifted_Ping_doPing, shifted_Ping_doStop;
    )
    (| shifted_Stop := Stop
     | shifted_Ping_doStart := shift_event(call_Ping_doStart , when (upsample_counter = nb_inserted))
     | shifted_Ping_doPing :=  shift_event(call_Ping_doPing , when (upsample_counter = (nb_inserted - 1)))
     | shifted_Ping_doStop :=  shift_event(call_Ping_doStop , when (upsample_counter = (nb_inserted - 2)))
     |);
```

Figure 22 Processus *unfold*

Le processus *putMsg*

Ce processus (cf. Figure 23~~Figure 23~~) n'écrit pas d'information dans la boite de réception, son rôle est seulement de créer une structure de données de type *Ping*_message pouvant être stockée dans la boîte de réception.

```
process putMsg =
    ( ? event Stop, call_Ping_doStart, call_Ping_doPing, call_Ping_doStop;
        integer param;
      ! Ping_message msg;
     )
    (| msg.id := (#Stop when Stop)
                 default (#call_Ping_doStart when call_Ping_doStart )
                 default (#call_Ping_doPing  when call_Ping_doPing )
                 default (#call_Ping_doStop  when call_Ping_doStop )
     | msg.deadline := 0
     | msg.param := (param when call_Ping_doStart) default 0
     |);
```

Figure 23 Le processus *putMsg*

Le processus *getMsg*

De même que le processus précédent, *getMsg* (Figure 24~~Figure 24~~) n'a aucune incidence sur la boîte de réception. Il est chargé de prendre un message déjà extrait de la boîte de réception et d'en extraire le type d'évènement, ses paramètres éventuels, ou encore l'appelant dans le cas des services avec valeur de retour.

```
process getMsg =
    ( ? Ping_message msg;
        boolean OK_get;
      ! event provided_Stop, provided_Ping_doStart, provided_Ping_doPing, provided_Ping_doStop;
        integer provided_doStart_value;
    )
    (| provided_Stop        := when (msg.id = #Stop) when OK_get
     | provided_Ping_doStart := when (msg.id = #call_Ping_doStart) when OK_get
     | provided_doStart_value := msg.param when (msg.id = #call_Ping_doStart) when OK_get
     | provided_Ping_doPing  := when (msg.id = #call_Ping_doPing ) when OK_get
     | provided_Ping_doStop  := when (msg.id = #call_Ping_doStop ) when OK_get
     |);
```

Figure 24 Le processus *getMsg*

A l'horloge précisée par *OK_get*, le message donné en entrée de ce processus est lu de façon à fournir au processus chargé de consommer les évènements, le prochain évènement à consommer ainsi que ses paramètres éventuels.

Le processus *FIFO*

Ce processus gère à la fois l'écriture et la lecture des messages dans la FIFO. Il fait l'objet d'un module SIGNAL, qui ne sera pas décrit ici en détail. Il nous importe juste de savoir :

- Qu'il écrit *msg_in* dans la FIFO quand il reçoit ce message en entrée,
- Qu'il lit et extrait un message de la FIFO quand le signal *clk_getMsg* est présent

Attention extraire un message de la FIFO ne veut pas forcement dire que ce message est consommé, seulement, que le prochain message à être consommé est stocké dans *msg_out*.

En sortie de la FIFO, divers indicateurs concernant l'état de la file sont mis à jour (fifo vide, fifo pleine, écriture réussie…). Enfin cette implémentation SIGNAL d'une file d'attente FIFO, permet l'écriture et la lecture simultanée de messages dans la file.

Les évènements externes ou internes sont traités à l'identique et écrits sous la forme de messages dans la file d'attente de l'objet.

Enfin, le processus *exec_xxx_chart* gère la consommation et le rejet des messages ainsi que l'appel des différents services de l'objet (Figure 25~~Figure 25~~).

```
module Ping =

process exec_Ping_chart =

process Ping_chart =
  ( ? event Tick;
      event ms;
      event Stop, call_Ping_doStart, call_Ping_doPing, call_Ping_doStop;
      …
    ! event end_step;
      event trigger_Pong_doPong, … ;
  )
  (|(| case Ping_State in
            {#On}:
            (| Ping_futureState ::= (#Off when (Stop …
             | …
             |)
            {#Off}:
            (| …
             |)
            end
             | Ping_currentState ^= …
             |)
             | end_step := …
             |)
  where
    type Ping_states = enum (On, Off);
    …
  end;

process Ping_doStart =

process Ping_doStop =

process Ping_doPing =

use General;
use A_fifo;
use RequiredInterface;

end;
```

Figure 25 Module de la classe temps réel *Ping* (partie 2)

Les évènement sont consommés par le processus *xxx_chart* quand ceux-ci sont disponibles, cependant afin de vérifier l'hypothèse de pas RTC, les évènement fournis à ce processus ne sont rendus disponibles qu'après la fin du traitement de l'évènement précédent (exécution du service compris).

Le processus *xxx_chart (cf.* Figure 26~~Figure 26~~*)*

Il représente la traduction en SIGNAL de la machine à états UML représentant le comportement des classes temps réel. Les actions sont :

- Vérification de la possibilité de tirer une transition (et enregistrement)

- Exécution du service défini par la transition
- Mise à jour de l'état courant
- Mise à jour d'une horloge *end_Step* représentant la fin du pas RTC.

```
process Ping_chart =
  ( ? event Tick;
      event ms;
      event Stop, call_Ping_doStart, call_Ping_doPing, call_Ping_doStop;
      …
    ! event end_step;
      event trigger_Pong_doPong, … ;
  )
  (|(| case Ping_State in
         {#On}:
         (| Ping_futureState ::= (#Off when (Stop … ) default (#On when call_Ping_doPing) default #On
          | (On_end_Ping_doStop, … ) := Ping_doStop(Stop … )
          | (On_end_Ping_doPing, … ) := Ping_doPing(call_Ping_doPing)
          | On_ignored_events := call_Ping_doStart
          | end_transition_from_On := On_end_Ping_doPing ^+ On_end_Ping_doStop ^+ On_ignored_events
          |)
         {#Off}:
         (| Ping_futureState ::= (#On when call_Ping_doStart) default #Off
          | (Off_end_Ping_doStart, … ) := Ping_doStart(call_Ping_doStart, … )
          | Off_ignored_events := Stop ^+ call_Ping_doStop ^+ …
          | end_transition_from_Off := Off_end_Ping_doStart ^+ Off_ignored_events
          |)
         end
          | Ping_futureState ::= defaultvalue Ping_futureState $
          | Ping_futureState ^= Ping_newState ^= Ping_State
          | Ping_end_transition := end_transition_from_On ^+ end_transition_from_Off
          | …
          | Ping_currentState := Ping_State
          | Ping_currentState ^= Tick
          |)
          | end_step := Ping_end_transition
          |)
  where
    type Ping_states = enum (On, Off);
    …
  end;
```

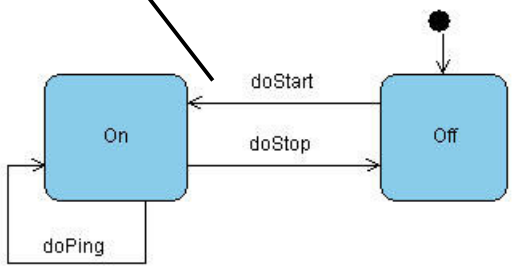

Figure 26 Le processus *Ping_chart*

Exécution d'un service

Si un service doit être appelé, cela correspond à l'appel du processus SIGNAL correspondant, appel qui doit mettre à jour une variable de fin d'exécution servant elle-même à mettre à jour la variable de fin de pas RTC. L'exemple ci-dessous donne un exemple de processus (traduction d'un service) pouvant être appelé

```
process Ping_doStart =
  ( ? event start_Ping_doStart;
      integer provided_doStart_value;
    ! event end_Ping_doStart;
      event trigger_Pong_doPong;
  )
  (| …
   | trigger_Pong_doPong := start_Ping_doStart
   | end_Ping_doStart := start_Ping_doStart
   |);
```

Figure 27 Un processus correspondant au service *doStart* de l'objet *Ping*

4.2 Naviguer dans le meta modèle UML avec OCL[9]

4.2.1 Pourquoi OCL ?

La première raison est bien évidemment le fait que l'on utilise UML comme langage de modélisation pour les modèles d'analyse. En effet, OCL est le langage mis en place par l'OMG pour :

- Parcourir le meta modèle UML
- Sélectionner des éléments dans ce meta modèle
- Vérifier des règles (contraintes) sur les éléments sélectionnés.

OCL n'est pas à proprement parler un langage de transformation de modèles (il ne permet pas de créer ou de modifier des modèles). Par contre, il est possible de décrire une transformation de modèles indirectement en jouant sur les pre et post conditions d'un ensemble d'opérateurs. Dans [19] il est proposé d'étendre UML avec des concepts permettant d'exécuter des action sur les modèles, de façon à éviter l'artifice qui consiste à utiliser des pre et post conditions, mais qui manque de lisibilité. Nous avons décidé de suivre cette approche qui sera décrite plus en détail dans le chapitre suivant.

La seconde raison est qu'il n'existe pas de langage de transformation de modèle qui fasse actuellement l'unanimité. Le plus général XSLT est un langage bas niveau qui n'est pas forcement adapté à UML. Les autres langages sont des langages propriétaires liés à des outils particuliers (TNI OTScript, Objecteering J, …).

4.2.2 Extensions pour la transformation de modèle

Pour décrire les transformations de modèles, il est nécessaires d'étendre le langage OCL avec des concepts de modification d'un modèle (création, suppression…). En effet, OCL est initialement prévu pour être un langage pour écrire des règles des contraintes ou des gardes dans les modèles. Ceci implique d'une part, que OCL est un langage parfaitement adapté au parcours du meta-modèle UML, mais d'autre part qu'il ne possède pas de mécanisme pour modifier le modèle parcouru.

L'action semantics en cours de standardisation à l'OMG, est apparue dans sa première version avec UML 1.4, afin de donner des mécanismes permettant de décrire des actions sur les modèles UML. Elle apporte les mécanismes qui manquent à OCL pour transformer des modèles. Actuellement, il n'existe de normalisée qu'une syntaxe abstraite de l'action semantics. Différentes implémentation sont proposées par des outilleurs (Action Specification Language, Action Language, Kabira Action Semantics[10]). L'utilisateur doit soit utiliser une syntaxe parmi celles qui existent, soit en définir une nouvelle.

Le choix fait dans le cadre du projet ACOTRIS est d'utiliser une version de OCL, étendue par un ensemble de concepts minimal mais suffisant pour la traduction que nous voulons décrire, issu de l'action semantics et auxquels nous donnons une sémantique concrète.

Pour être plus précis, il existe bien une version de UML avec « action semantics », mais toute la partie liée à l' « action semantics » ne propose qu'une syntaxe abstraite. Certains outils l'implémentent et associent une syntaxe concrète

Rappel du contexte :

- Lire un modèle UML
- Ecrire un modèle SIGNAL

Nous avons besoin :

- D'un langage pour parcourir le méta modèle UML : OCL
- De mécanismes pour créer des éléments du langage SIGNAL : AS extensions pour OCL

[9] Object Constraint Language,

[10] Différents langages d'action qui implémentent la sémantique sont décrits dans l'annexe B du document de spécification du langage OCL.

- D'un langage pour parcourir le modèle SIGNAL à créer : OCL

L'utilisation de l'OCL pour parcourir le modèle SIGNAL, est possible si l'on est à même de donner un meta-modèle syntaxique minimal, sous la forme d'un diagramme de classes, capable de décrire la syntaxe de n'importe quel modèle SIGNAL. Un meta-modèle satisfaisant ces conditions est décrit dans le chapitre suivant.

Extensions d'OCL pour la transformation de modèles :

- Affectations

Les affectations se font par le délimiteur « := ». Les actions sont terminées par le délimiteur « ; ».

Ex : a **:=** 'Hello' **;**

- Création d'objets

La création d'un objet est décrite à l'aide du constructeur « new » suivi du nom de la classe à instancier.

Ex : a := **new** ClasseName **;**

Si la classe possède plusieurs attributs, il est possible de les initialiser.

a := **new** ClassName **with**

Attr_1 **:=** value_1**,**

Attr_2 **:=** value_2**,**

…

Attr_n **:=** value_n **;**

Cette syntaxe est largement inspirée de l'ASL (Action Semantisc Specification) dont le manuel est disponible à l'adresse http://www.kc.com/.

- Relations entre objets

Une relation entre objets est instanciée par l'appel d'une méthode append_roleName, ou roleName est le nom du rôle de l'objet à rattacher sur la relation voulue.

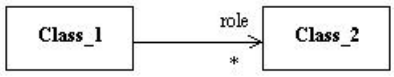

Figure 28 2 classes et 1 relation

Ex : O1 := new Class_1 ;

O2 := new Class_2 ;

O1→**append_role**(O2) ;

Dans le cas général de la transformation de modèles UML, nul doute que d'autres mécanismes seraient nécessaires (suppression d'éléments du modèle…). Nous ne décrivons ici que les mécanismes nécessaires et suffisants à la description de la transformation d'un modèle UML vers SIGNAL.

4.3 Un meta modèle syntaxique de SIGNAL décrit en UML

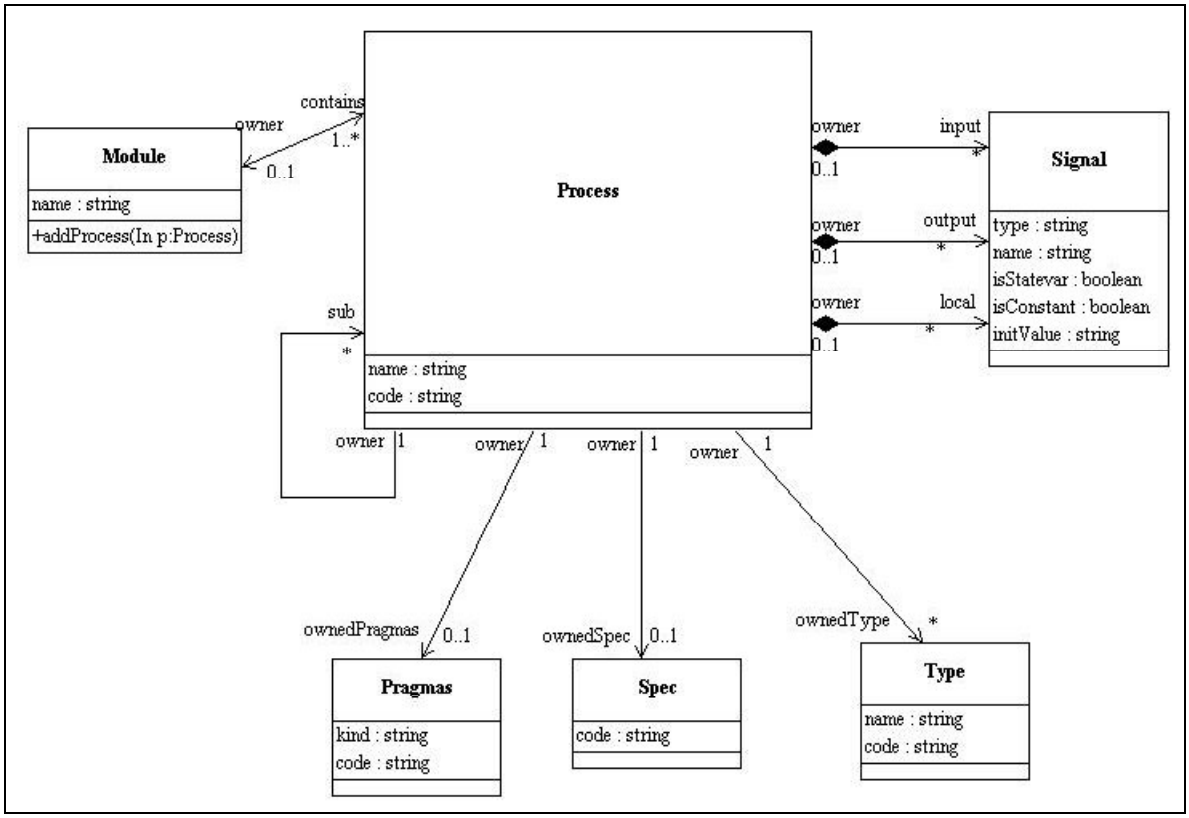

Figure 29 Un méta modèle syntaxique permettant l'écriture d'un programme en SIGNAL

Le méta modèle décrit ci-dessus ne décrit en aucun cas la sémantique complète de SIGNAL. Le but recherché ici est d'offrir un nombre minimum de concept de la syntaxe de SIGNAL, qui permette de décrire un programme SIGNAL. Ce meta modèle nous sert à décrire la transformation UML vers SIGNAL :

- En parcourant le modèle UML et en sélectionnant des éléments à l'aide de règles OCL.
- En créant, des éléments à l'aide d'actions dans un nouveau modèle, instance du meta-modèle syntaxique SIGNAL défini précédemment.

Globalement, on manipule deux référentiels, UML et un Meta Modèle SIGNAL. Lecture et sélection se font sur le meta modèle UML source, l'écriture et les modifications portent sur le modèle SIGNAL cible.

4.4 Formalisation

4.4.1 Le processus global

4.4.2 Les OTR

4.4.3 Classes simples

Par classe simple on entend toutes les classes du cœur du système (on ne considère donc pas les interfaces) qui ne sont pas des objets temps réels.

Une telle classe (Figure 30~~Figure 30~~) est traduite en SIGNAL par un module qui regroupe sous la forme de processus SIGNAL, l'ensemble des opérations détenues qu'elle contient.

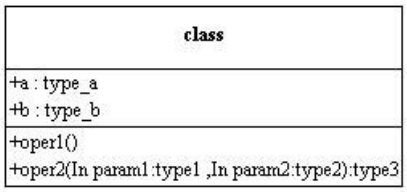

Figure 30 classe simple du modèle

```
module class =

process class_oper1 =
  ( ?       event clk_oper1;
    !
  )
  (|
   |);

process class_oper2 =
  ( ?       event clk_oper2;
            type1 param1;
            type2 param2;
    !       type3 out_oper2;
  )
  (|
   |);

end;
```

Figure 31 Traduction signal d'une classe simple

La déclaration des paramètres doit manipuler des types compréhensibles par SIGNAL, c'est à dire des types simples (integer, short, string,…). Dans le cas de l'utilisation de types plus complexes, il faudra définir ceux–ci en SIGNAL. Les attributs a et b de la classe de la Figure 30~~Figure 30~~, ne sont pas traduits dans le module qui représente la classe, mais sous la forme de variable « statevar[11] » dans le processus principal.

[11] Une variable dite "statevar " est en SIGNAL, une variable définie à l'horloge la plus rapide du système, elle est définie au niveau du processus principal de façon à être accessible depuis n'importe quel processus du programme SIGNAL généré.

```
context Class::createSimpleModule : Module
pre :
        not(self.stereotype.name = "RealTimeObject")
actions : -- Signal MM context
        m := new Module with name := self.name;
        self.featureOperation->iterate(o : Operation | -- parcours des operations
                p := new Process with name := o.name;
                p.addInput("event", String("clk_").concat(o.name));
                o.parameter->iterate( param : Parameter |
                        if (param.kind = "return")
                        then
                                -- traitement particulier si valeur de retour
                                p.addOutput(param.type.name, String("out_").concat(param.name));
                        elseif (param.kind = "in")
                        then
                                -- les paramètres sont des entrées
                                p.addInput(param.type.name, param.name);
                        else
                                -- case param.kind = out
                                -- interdit
                                -- case param.kind = inout
                                -- interdit
                        endif
                )
        )
        result := m

context Process::addInput(type_name : String, input_name : String)
pre:
actions:
        s := new Signal with
                type := type_name,
                name := input_name;
        self.append_input(s);

context Process::addOutput(type_name : String, output_name : String)
pre:
actions:
        s := new Signal with
                type := type_name,
                name := output_name;
        self.append_output(s);
```

Figure 32 Traduction OCL étendu pour une classe simple

```
context
```

4.4.4 Les interfaces requises

5 CONCLUSION

Bibliographie

1. OMG, *UML Specifications 1.4*. 2001, OMG.
2. Gérard, S., et al., *Methodology for developing real time embedded systems*. 2002, CEE: Paris. p. 158.
3. Gérard, S., *The ACCORD/UML profile*. 2002, CEA - LIST: GIF sur YVETTE.
4. Guernic, L., et al. *Programming Real-Time Applications with Signal*. in *Proceedings of the IEEE*. 1991.
5. Grandpierre, T., C. Lavarenne, and Y. Sorel. *Optimized Rapid Prototyping For Real Time Embedded Heterogeneous Multiprocessors*. in *CODES'99 7th International Workshop on Hardware/Software Co-Design*. 1999. Rome.
6. Eshuis, R. and R. Wieringa. *Requirements-level semantics for UML statecharts*. in *FMOODS 2000*. 2000: Kluwer.
7. Lilius, J. and I. Porres. *Formalizing UML state machines for model checking*. in *UML'99*. 1999: Springer Verlag.
8. Latella, D., I. Majzik, and M. Massink. *Towards a formal operational semantics of UML statechart diagrams*. in *FMOODS'99, IFIP*. 1999: Kluwer.
9. Reggio, G., et al. *Analysing UML active classes and associated state machines – a lightweight formal approach*. in *FASE 2000*. 2000: Springer-Verlag.
10. Yonezawa, A., *ABCL: An object-oriented concurrent system*. 90: MIT Press.
11. Terrier, F., D. Bras, and P. Vanuxeem. *Intelligent Real Time Control: Real Time Scheduling Design in Object Oriented Approach*. in *Workshop on Object-Oriented Real-Time Systems Analysis and Design Issues, OOPSLA'93*. 93. Washington, DC, USA.
12. Takashio, K. and M. Tokoro. *DROL: An object-oriented programming language for distributed real-time systems*. in *The Object-Oriented Programming : Systems, Languages and Applications Conference (OOPSLA'92)*. 92: ACM SIGPLAN Notices.
13. Kafura, D., M. Mukherji, and G. Lavender, *ACT++: a class library for concurrent programming in C++ using actors.* Journal of Object-Oriented Programming (JOOP), 93(Octobre): p. 47-55.
14. OMG, *Meta Object Facility Specifications1.4*. 2002, OMG.
15. André, C., M.-A. Péraldi-Frati, and J.-P. Rigault, *UML et le paradigme synchrone : Application à la conception de contrôleurs embarqués*. 2002, Laboratoire I3S.
16. Soley, R. and t.O.S.S. Group, *Model Driven Architecture (Draft 3.2)*. 2000, OMG. p. 8.
17. SOFTEAM, *UML Profiles and the J language*. 1999, SOFTEAM.
18. Gérard, S., et al., *UML based methodology for real time embedded systems*. 2003, AIT-WOODDES (IST-1999-10069).
19. Pollet, D., D. Vojtisek, and J.-M. Jézéquel. *OCL as a Core UML Transformation Language*. in *WITUML 2002 - Position Paper*. 2002. Malaga, Spain.