Contribution à la flexibilité et à la rapidité de conception des systèmes automatisés avec l'utilisation d'UML

Fabien Chiron

U.F.R. Sciences et Technologies de Clermont-Ferrand
École Doctorale Sciences Pour l'Ingénieur

THÈSE

Réalisée en vue de l'obtention du titre de

Docteur en Sciences

de L'université Blaise Pascal de Clermont-Ferrand

Spécialité : *Informatique*
Orientations : *Automatisme*, *Informatique Industrielle*

Fabien CHIRON

Contribution à la flexibilité et à la rapidité de conception des systèmes automatisés avec l'utilisation d'UML

Présentée devant le Jury:
Président: Eric NIEL - INSA Lyon
Rapporteur: Jean-Jacques LESAGE – ENS Cachan
Rapporteur: Jean-François PETIN – ESIAL Nancy
Examinateur: Didier NOTERMAN – INSA Lyon
Examinateur: Guy SOUCHET – Newtec Case Palletizing
Directeur de Thèse: Alain QUILLOT - LIMOS Clermont-Ferrand
Co-Directeur de Thèse: Khalid KOUISS – LIMOS Clermont-Ferrand

Réalisée en collaboration avec le Laboratoire d'Informatique, de Modélisation et d'Optimisation des Systèmes (LIMOS), l'Institut Français de Mécanique Avancée (IFMA) et l'entreprise Newtec Case Palletizing (NCP).

À mes proches...

« *C’est en parvenant à nos fins par l’effort, en étant prêts à faire le sacrifice de profits immédiats en faveur du bien-être d’autrui à long terme, que nous parviendrons au bonheur caractérisé par la paix et le contentement authentique.* »

Tenzin Gyatso (14ème Dalaï-Lama)

« *L'important n'est pas de convaincre, mais de donner à réfléchir* »

Bernard Werber ('*Le père de nos pères*', '*L'encyclopédie du Savoir Relatif et Absolu*')

Remerciements

Ils vont en premier lieu à mon co-directeur de thèse Khalid KOUISS, pour sa gentillesse et sa patience, son soutien et ses encouragements continuels qui ont permis l'aboutissement de ces efforts en dépit des nombreuses difficultés rencontrées dans les phases successives de ce travail. Il m'a aidé à gérer les facettes scientifiques et industrielles inhérentes aux contrats CIFRE, ce qui a contribué à préparer mon insertion professionnelle. Je remercie également Alain QUILLIOT d'avoir co-encadré cette thèse et autorisé ma collaboration avec Khalid.

Ensuite je tiens à exprimer ma reconnaissance chaleureuse à Guy SOUCHET, responsable R&D Automatisme de l'entreprise partenaire, dont l'écoute ouverte et critique, l'expérience concrète du terrain, m'ont permis de réaliser un travail pragmatique.

Je remercie Jean-François PETIN, d'avoir été le rapporteur de cette thèse, de sa lecture attentive et critique, ainsi que Jean-Jacques LESAGE qui, bien que ne m'ayant pas rencontré avant la soutenance, a également apporté des remarques pertinentes sur les propositions de ce manuscrit qu'il a patiemment disséqué. Je salue également l'intérêt porté par Didier NOTERMAN et Éric NIEL, qui ont accepté de se joindre au jury pour évaluer l'originalité du travail.

Cette thèse a été lancée dans le cadre d'un projet de recherche dont je remercie d'autres initiateurs, Pierre GUASCO et Alain CONTINI qui m'ont fait confiance en donnant leur accord pour cette aventure, puis Yann DELVAUX et Laurent FERRE qui, dans un contexte économique difficile ont soutenu mes efforts en me laissant une indépendance scientifique et temporelle.

Je remercie également Michel JAMIN et les membres du bureau d'étude automatisme dont il est responsable, pour leur accueil, leur disponibilité, notamment Pascal HERVE.

Enfin, ma gratitude va à mon entourage et son soutien sans faille, Emmanuelle, Geneviève, Maurice, Nicolas et les autres, avec un merci particulier à Jacques et Marie-Joseph pour leur maîtrise de la langue de Molière.

RÉSUMÉ

La dynamique actuelle des marchés entraîne avec elle une complexité croissante des demandes du client et nécessairement des contraintes de production. Les méthodologies traditionnelles de conception de systèmes montrent leurs limites dans des contextes très changeants pour lesquels les spécifications sont amenées à évoluer rapidement, des éléments technologiques particuliers de réalisation étant souvent pris en compte trop tôt dans le travail d'étude, limitant la versatilité des développements. Les entreprises doivent alors capitaliser au maximum les efforts menés dans les phases amont de spécification pour optimiser les temps d'étude.

Notre travail de recherche s'intéresse plus précisément au domaine des systèmes automatisés et se propose de répondre à la problématique précédente en utilisant des techniques issues du monde de l'informatique pour la réalisation des systèmes physiques, comme l'**OOA** (*Approche Orientée Objet*) et la modélisation objet **UML** (*Langage de Modélisation Unifié*) avec la perspective d'une spécialisation tardive et d'une génération automatique selon les cibles technologiques choisies, comme le préconise la logique **IDM** (*Ingénierie Dirigée par les Modèles*).

L'originalité de ce mémoire est de décrire une méthodologie et une organisation de travail pour la conception des systèmes automatisés, en s'appuyant sur le concept d'objet d'automatisme multi-facettes. De plus, nous proposons une utilisation de l'extension **SysML** (*Langage de Modélisation des Systèmes*) pour la représentation d'éléments d'automatisme particuliers, les *blocs fonctions* de la norme **IEC 61131-3**, à travers le stéréotype « *block* ». Enfin nous montrons comment il est possible d'obtenir une première génération de code automate en passant par les spécifications **PLCopen**, définissant un lien entre une syntaxe **XML** (*Langage de balisage eXtensible*), se voulant standard, et les langages de la norme **IEC 61131-3**. Le passage par cette représentation standardisée permet de garder l'indépendance des implémentations vis-à-vis d'un environnement intégré de développement particulier.

Le processus de conception décrit a été appliqué à un cas d'étude industriel réel appartenant au domaine de la palettisation robotisée.

ABSTRACT

The actual economical market tendency brings an increasingly complexity of customer needs and consequently, of the production constraints. The current design methodologies show their limits in very inconstant contexts where specifications have to rapidly evolve, due to too early consideration of specific technological items in the preliminary specification work, limiting developments versatility. Then companies have to capitalize the leaded efforts in first specification phases in order to reduce development time.

Our thesis work deals with the Automation Systems Engineering and intends to answer the previous issue using techniques from software design areas to specify physical systems, like the **OOA** (*Object-Oriented Design*) and object-oriented modeling with **UML** (*Unified Modeling Language*), targeting a late specialization and a code generation according to the chosen technological target, like in the **MDA** (*Model Driven Architecture*).

The original work here is in the description of a methodology and a working organization for Control and Automation systems, according to a multi-aspect automation object. Furthermore, we propose the use of **SysML** (*Systems Modeling Language*) to represent the *function blocs* in the **IEC 61131-3** standard, through the « *block* » stereotype. Finally, we show how it is possible to obtain a first code generation using the PLCopen specifications, defining a link between a **XML** description(*eXtensible Markup Language*), aiming at being a standard, and the **IEC 61131-3** norm languages. Using this, independence from a specific integrated development environment is kept.

The described design process has been transposed to a real industrial case study, in robotic palletization area.

Table des Matières

Liste des Définitions

Index des Figures

CHAPITRE 1 :

INTRODUCTION

1.1 Contexte des travaux

Ce travail de recherche a été mis en oeuvre dans le cadre d'un **contrat CIFRE** (*Convention Industrielle de Formation par la Recherche*) n°270/2004 du 01/08/2004 sponsorisé par l'**ANRT** (Association Nationale pour la Recherche). Ces conventions permettent à des entreprises de recruter des jeunes doctorants dont le travail de recherche et de développement doit aboutir à la soutenance d'une thèse de doctorat. En l'occurrence, l'entreprise bénéficiaire **Newtec Case Palletizing** est spécialisée dans la construction de machines de conditionnement, d'emballage et de manutention continue. Le laboratoire partenaire est le **LIMOS** (*Laboratoire d'Informatique, de Modélisation et d'Optimisation des Systèmes*) de Clermont-Ferrand.

Les entités ciblées par cette étude sont les lignes de production automatisées. Les éléments en amont peuvent être: de la matière première, des éléments semi-finis. Ils sont transformés ou organisés au cours d'un processus en général complexe aboutissant à la réalisation de produits finis ou semi-finis prêts à être expédiés. Le rythme de production de ces installations dans notre étude est très élevé (milliers d'unités par heure) et la main d'oeuvre n'est pas à même de réaliser des opérations élémentaires à la vitesse

nécessaire. L'introduction d'éléments d'automatisme permet d'effectuer ces tâches de façon répétitive dans un laps de temps dont l'incertitude est déterminée. Par contre le système devient plus complexe puisqu'on introduit des réseaux de communication, des programmes de commande, des éléments opérationnels avec des interactions qu'il faut pouvoir maîtriser.

Au cours de ces trois années j'ai été rattaché au Bureau d'Études Automatisme de l'entreprise ce qui m'a permis de collaborer avec des spécialistes de l'automatisme et de valider sur des prototypes concrets les outils et les approches développés au cours de cette thèse. La principale difficulté a résidé dans le fait de concilier des objectifs industriels à court terme liés aux résultats de mes travaux avec le recul et la rigueur nécessaires à la démarche de recherche du travail de thèse.

Au cours de la première moitié de mon activité j'ai également pris part de façon active au projet **CLIPS** (*Composants Logiciels pour Îlots de Palettisation robotiséS*) lancé début 2004 et qui répondait à un appel à projets **RNTL** (*Réseau National de recherche et d'innovation en Technologies Logicielles*). J'ai pu notamment échanger et travailler avec le **LIST** (*Laboratoire d'Intégration des Systèmes et des Technologies*) et le **CEA** (*Commissariat à l'Énergie Atomique*). Un ensemble de réflexions, de concepts, de développements logiciels et d'outils a été proposé pour la mise en oeuvre de l'approche par composants pour le développement d'applications de contrôle/commande distribuées, avec la volonté d'appliquer le paradigme de la programmation par composants au développement d'outils logiciels pour l'industrie manufacturière.

1.2 Un peu d'histoire

Pour bien appréhender les problématiques spécifiques développées dans ce mémoire, il est important de prendre du recul par rapport à l'industrie en général et d'analyser l'évolution des méthodes de travail et du marché pour bien comprendre les tendances, les enjeux actuels à un niveau plus général.

Il y a cinquante ans, la demande du marché en biens manufacturés était très élevée et tout ce qui était produit trouvait immédiatement un acheteur. Les exigences du

client étaient simples, il fallait beaucoup et très vite. On développa donc des systèmes spécialisés dans l'élaboration continue de produits particuliers et tous les efforts de développement furent portés sur l'augmentation de la capacité de production souvent par multiplication des unités élémentaires de réalisation. La production de masse, le Taylorisme [Taylor 1911] et une certaine vision de l'**O.S.T.**(*Organisation Scientifique du Travail*) constituaient à cette époque le credo des investisseurs et des industriels.

La technique évoluant, les processus devinrent plus élaborés et nécessitèrent un enchaînement d'opérations interdépendantes réalisées à l'aide de machines possédant des capacités différentes. Le flux devint discontinu et le rythme de production diminua; on multiplia les tâches les plus lentes pour maintenir une utilisation maximale de toutes les ressources. Le débit fut maintenu mais les investissements devinrent plus importants.

Avec les crises pétrolières et, de façon induite, sociales, l'inéquation demandes-besoins s'inverse brusquement, l'émergence sur le marché de la concurrence mondiale se fait ressentir de plus en plus lourdement, les prix de vente chutent, les méthodes de production traditionnelles deviennent financièrement inadaptées à la survie des industries. Beaucoup se tournent alors vers le Japon, et observent ces entreprises qui réagissent parfaitement à la crise comme la firme Toyota, mettant en évidence de nouveaux schémas d'organisation du travail comme le toyotisme [Ohno 1995] (Juste à Temps, Kanban, Flux tiré), l'automatisation des processus est à l'honneur. On souligne l'importance de la réactivité, c'est à dire de la capacité d'un système à réagir rapidement à l'évolution des ordres de production, de façon flexible et autonome.

Aujourd'hui, le marché est complètement changeant. Il faut que les produits soient personnalisables, variés et peu chers pour séduire le client, et la concurrence mondiale est particulièrement présente. La réactivité n'est plus un gage de survie, il faut sans arrêt innover et pouvoir mettre en place de nouveaux produits avec les moyens de production correspondants dans des laps de temps très réduits. Il est alors nécessaire de capitaliser les efforts passés de façon structurée pour ne consacrer du temps que sur les éléments réellement nouveaux. La phase de conception devient une composante essentielle dans l'évaluation de l'efficacité d'une structure flexible de production.

1.3 **Problématique et domaine d'étude**

Cette thèse se place dans une problématique induite par les tendances actuelles en matière d'organisation du travail, à savoir comment augmenter l'efficacité des phases de conception pour qu'elles puissent d'une part permettre des mises en oeuvre plus rapides au niveau de la production en aval, et d'autre part être capitalisées pour une réutilisation future.

Mettre à disposition de nouveaux moyens de production rapidement, c'est accélérer les phases qui composent leurs cycles de développement, de la spécification à la mise en service. Favoriser la création et la réutilisation des outils et des applications mis en place lors de projets précédents devient alors une condition indispensable à l'efficacité et à la rapidité de réalisation des nouveaux produits.

Une définition simplifiée de la conception de systèmes en rapport avec le milieu industriel pourrait être :

DÉFINITION 1 : CONCEPTION

La conception est le processus qui permet à l'entreprise de passer d'une idée de produit répondant à un besoin, à sa concrétisation physique, en définissant également les moyens de sa fabrication.

Pour les entreprises localisées dans les pays dits « *riches* », la logique de compétitivité « *par les prix* » n'est plus viable face à la concurrence des pays émergents et ce, malgré les efforts accomplis en terme de productivité durant les dernières décennies; la compétitivité par une excellence en conception devient une réelle alternative pertinente pour l'industrie comme souligné dans ce rapport du ministère [RF 2003]. On préfère d'ailleurs de plus en plus l'appellation « *créateur* » à celle de « *producteur* » dans les services marketing des grandes compagnies (notamment dans l'automobile), ce qui reflète également une réalité en terme de volonté d'investissement dans l'innovation et la **R&D** (*Recherche et Développement*) pour ces entreprises.

DÉFINITION 2 : INNOVATION [OCDE 1992]

L'innovation de produit se caractérise par « l'introduction sur le marché d'un produit (bien ou service) nouveau ou nettement modifié au regard de ses caractéristiques fondamentales, de ses spécifications techniques, des logiciels incorporés ou de tout autre composant immatériel, ainsi que de l'utilisation prévue ou de la facilité d'usage ».

L'innovation de procédé se définit par « l'introduction dans l'entreprise d'un procédé de production, d'une méthode de fourniture de services ou de livraison de produits, nouveaux ou nettement modifiés. Le résultat doit être significatif en ce qui concerne le niveau de production, la qualité des produits ou les coûts de production et de distribution ».

Le découpage des fonctionnalités en composants semble répondre de manière adéquate à cette problématique, introduisant une flexibilité non seulement dans la mise en œuvre de la phase de construction d'un système mais également dans sa maintenabilité et son évolution possible au cours du temps.

Beaucoup de travaux de recherche ont déjà été menés dans ce domaine notamment pour des applications informatiques. L'objet de cette thèse est d'élaborer une méthodologie de conception plus globale s'appuyant sur le paradigme componentiel pour les différentes phases de la mise en œuvre d'un projet industriel concret de conception d'un système automatisé.

Les nouvelles représentations du langage de modélisation **UML** (*Unified Modeling Language*) dans sa version (2.1) permettent d'enrichir la description de composants qui modélisent toutes les facettes du développement d'un produit; on peut ainsi intégrer au sein d'un même composant des aspects qui étaient précédemment développés séparément. L'encapsulation des parties mécaniques, électroniques, d'automatisme, de supervision et de détection d'anomalies au sein d'une même entité de conception, offre la possibilité de réutiliser et d'intégrer facilement des modules fonctionnels vus sous forme de boîtes noires et dont l'instanciation dans un projet entraîne le déploiement des caractéristiques du composant dans les domaines précédemment cités, facilitant le travail des différents intervenants spécialistes d'un aspect de ce déploiement.

Le développement d'un outil d'aide à la conception sera mis en place pour la construction d'îlots de palettisation robotisés. Ensuite, une partie « *synthèse de commande* » devra permettre la génération et l'intégration dans les ateliers logiciels spécifiques à une activité donnée (mécanique, automatisme, électronique, informatique) de la partie correspondante au sein du composant. Le format d'échange extensible **XML** (*Extensible Markup Language*) permettra l'importation et l'enregistrement des aspects correspondants pour un composant donné.

Disposant d'une bibliothèque de modules métiers dont le déploiement et l'assemblage sont facilités, et ayant la possibilité d'enrichir cette librairie en suivant une méthodologie adaptée, l'entreprise bénéficiaire pourra concevoir de nouveaux ensembles de façon plus saine et plus rapide et estimer le prix de la mise en œuvre de ces systèmes plus ou moins complexes, en fonction de la qualité et de la richesse de fonctionnement recherchées.

Ce travail s'effectuant dans un cadre industriel spécifique lié au métier de la palettisation, nous prendrons comme contexte d'étude celui des systèmes automatisés.

1.4 Plan de la thèse

Nous allons dans un premier temps introduire le domaine particulier dans lequel nous situons nos travaux, les systèmes à base d'automatisme. Nous présenterons les caractéristiques de ces structures avec la façon dont on les conçoit et dont on implémente toute la logique qui permet de les contrôler ainsi que les langages spécifiques qui permettent de le faire. Ces considérations feront l'objet du Chapitre 2.

Dans le Chapitre 3 nous décrivons les langages de programmation actuellement utilisés par les automaticiens afin d'en comprendre les principales caractéristiques et d'introduire les correspondances que nous établirons dans le chapitre 7 avec la présentation de notre approche méthodologique.

La modélisation fait aujourd'hui partie intégrante des processus de conception, cette constatation est détaillées dans le Chapitre 4 avec un passage en revue des principales méthodologies d'analyse fonctionnelle et systémique existantes pouvant s'appliquer aux systèmes de production automatisés.

Face à la complexité croissante des systèmes, les approches traditionnelles sont en revanche peu efficaces lorsque le système et ses spécifications doivent évoluer au cours du temps. L'approche orientée objet pour la conception est une réponse pertinente à ce type de problématique, elle fait l'objet du Chapitre 5.

Nous aurons alors vu tout l'intérêt de la modélisation et de l'approche objet, nous enchaînerons naturellement avec la présentation du langage phare en terme de représentation objet, **UML** (*Unified Modeling Language*), dans le Chapitre 6 et des travaux de l'**OMG** (*Object Management Group*) sur le concept plus général de l'architecture **MDA** (*Model Driven Architecture*).

Le Chapitre 7 souligne les insuffisances du langage **UML** non étendu dans le cadre des systèmes automatisés et présente les travaux ayant été effectués sur la correspondance entre langages automates et des extensions possibles d'**UML**. L'apport original porte sur l'utilisation de l'extension **SysML** (*System Modeling Language*), avec l'optique d'une génération automatique de code, au sein d'une méthode basée sur un concept de composant multi-facettes pour les systèmes à base d'automatisme.

Nous présentons à la fin du chapitre le détail de cette approche et proposons une méthodologie de spécification et de déploiement pour les systèmes automatisés en utilisant la modélisation.

Des perspectives pour des travaux ultérieurs sont présentées dans le chapitre 8 avec une conclusion sur le travail. Enfin, le Chapitre 9 évoque succinctement le matériel qui a servi de support concret d'étude pour cette thèse.

CHAPITRE 2 :
LES SYSTÈMES AUTOMATISÉS

Dans ce chapitre nous introduisons les systèmes automatisés, leurs différentes architectures de fonctionnement et les technologies qui leur sont associées. Nous décrivons également la façon dont on conçoit traditionnellement ces installations, notamment au niveau de la réalisation des logiques de fonctionnement.

2.1 **Caractéristiques**

Bien que l'histoire de l'automatique remonte aux premiers systèmes ayant permis de réguler des actions physiques, les systèmes automatisés ont vraiment montré leur intérêt avec l'apparition de l'électronique durant la première moitié du XXe siècle, permettant de traduire des grandeurs physiques en signaux électriques pour leur appliquer des opérations mathématiques par la suite; ainsi retrouve-t-on une excellente et très instructive étude sur l'histoire de l'automatisme dans [Remaud 2004]. Aujourd'hui, ils sont indissociables des systèmes de production modernes même s'ils souffrent d'une très

mauvaise réputation d'un point de vue social puisque leur intégration correspond généralement au remplacement d'activités précédemment assurées par l'homme.

DÉFINITION 3 : SYSTÈMES AUTOMATISÉS DE PRODUCTION (SAP)

Un système automatisé de production est constitué d'une Partie Commande et d'une Partie Opérative. La partie commande contient la logique de fonctionnement du processus que l'on veut automatiser, elle envoie des ordres à la partie opérative qui les exécute à l'aide d'actionneurs et qui lui renvoie des informations à partir de capteurs. La partie commande gère également le dialogue avec l'opérateur par l'intermédiaire le plus souvent d'une Supervision.

Sur la Figure 2.1 on retrouve ces trois principaux niveaux de contrôle avec les catégories d'informations qui vont circuler entre eux.

Dans nos travaux nous présentons une approche de conception englobant ces aspects et permettant de faciliter la génération d'une logique de contrôle selon la cible de déploiement. L'élément central et le plus spécifique de ces systèmes est la partie commande traditionnellement constituée d'un Automate Programmable Industriel (**API**), que l'on retrouve également beaucoup sous l'acronyme anglais **PLC** (Programmable Logical Controller), et qui est une plate-forme d'exécution particulière, spécialement conçue pour répondre aux besoins et aux contraintes de l'architecture de contrôle.

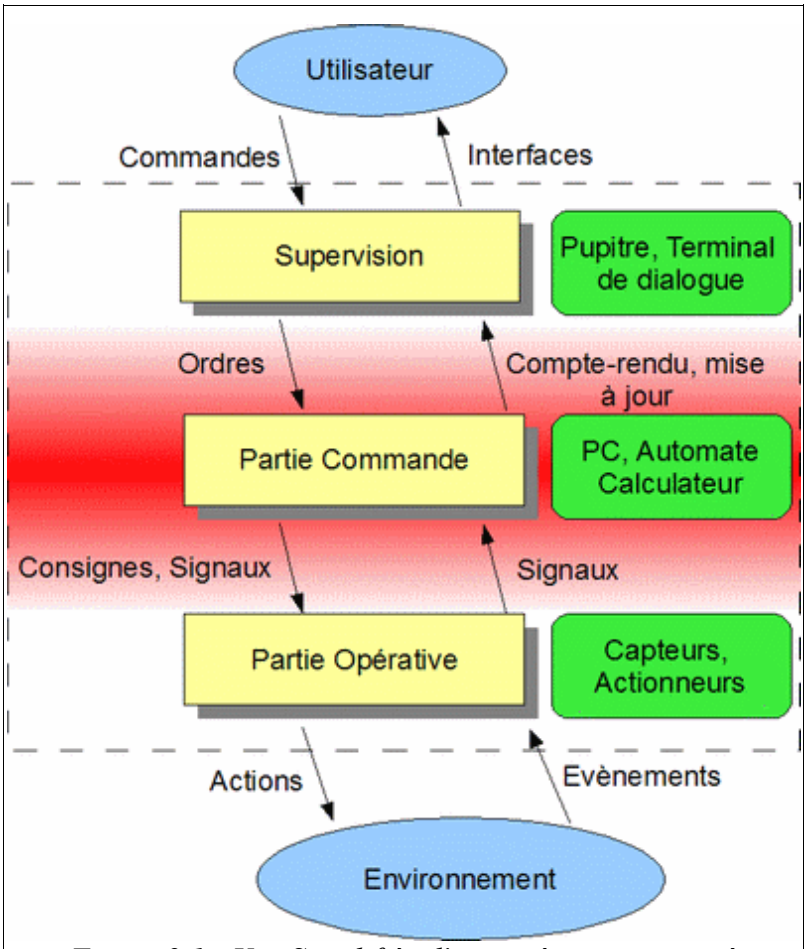


Figure 2.1.: Vue Simplifiée d'un système automatisé

2.1.1 Partie Supervision

Les deux composantes principales de cette partie concernent le suivi et le pilotage du processus industriel.

Le suivi se fait généralement par l'intermédiaire d'une interface graphique ou d'un pupitre de commande déportés physiquement ou directement présents au sein de l'installation. Cette fonction de suivi permet de vérifier le bon fonctionnement d'un équipement ou le bon déroulement d'un procédé. En fonctionnement normal, la supervision permet la visualisation d'indicateurs visuels ou la diffusion vers l'extérieur d'informations liées à l'évolution du système pour la sauvegarde ou le traitement

immédiat. Lorsque le fonctionnement devient anormal, le rôle de la supervision est d'aider l'opérateur à ramener le système dans un état de fonctionnement nominal.

La supervision doit détecter les défaillances éventuelles d'un système, et même les anticiper si elle possède des modèles de comportement permettant de prévoir certains enchaînements d'évènements, puis de signaler les défauts à des personnes ou à des systèmes informationnels de niveau supérieur.

La détection se compose en général d'une description de l'anomalie, notamment sa nature et sa localisation, puis de la présentation des actions à réaliser pour corriger l'erreur en cours et redémarrer correctement le déroulement normal dynamique du système concerné.

Lorsque le système se trouve dans des phases intermédiaires de fonctionnement (arrêt, mode dégradé, défaut), la supervision permet également le pilotage des éléments physiques et le démarrage de cycles spéciaux ou automatiques.

Dans notre étude, nous considérerons que la supervision est implémentée par l'intermédiaire d'un ensemble d'applicatifs informatiques communiquant entre eux et s'interfaçant à la fois avec la partie commande et les utilisateurs du système.

La plate-forme d'exécution qui accueille la supervision pour le contrôle d'automatismes est en général un matériel informatique dédié relié à un écran de contrôle.

2.1.2 Partie Commande

Les systèmes de commande gèrent des installations dans lesquelles ils doivent assurer l'intégrité des personnes physiques et des équipements. Ils ont l'obligation de réagir aux évolutions de l'environnement dans un laps de temps donné et contrôlé. Certaines installations peuvent avoir des conditions d'utilisation très difficiles (poussière environnante, perturbations électromagnétiques, vibrations, variations de température). Ces contraintes ont entraîné la construction d'éléments d'exécution très robustes constitués de circuits électroniques. D'abord simples calculateurs sans processeur ni mémoire, les systèmes de commande ont ensuite évolué vers des matériels toujours aussi robustes mais programmables avec mémoire vive, les **APIs**. Le déterminisme temporel et

le degré de criticité élevé des systèmes où ils sont implantés ont amené la spécification de langages particuliers par l' **IEC** (*International Electrotecnical Commission*) et destinés à une logique d'exécution séquentielle et déterministe.

Nous détaillerons un peu plus loin ces différents langages et leurs caractéristiques.

2.1.3 Partie Opérative

C'est la partie opérationnelle de l'architecture avec des éléments mécaniques, pneumatiques, hydrauliques que l'on peut généraliser comme étant des actionneurs et répondant aux signaux électriques envoyés par la partie commande, réalisant ensuite physiquement le processus industriel. On retrouve également des éléments de détection physique comme les capteurs qui permettent de renseigner la partie commande sur l'état du système à un instant donné, de façon passive (c'est alors la partie commande qui se charge d'aller vérifier l'état du capteur) ou de façon active (déclenchement d'un signal lors d'un événement physique particulier).

Les signaux échangés transitent sur des réseaux de communications avec pléthore de protocoles selon la technologie choisie.

2.2 **L'API, élément central de l'architecture**

2.2.1 Vue Générale

Pour bien comprendre les particularités de la programmation que nous allons étudier plus loin, il est important de revenir sur la nature même de fonctionnement des **APIs**. Sur la Figure 2.3, nous avons représenté la structure de base d'un **API** avec les différents éléments internes qui le composent. On retrouve une architecture matérielle typique des dispositifs embarqués, avec un microprocesseur, une mémoire, des circuits d'entrée et de sortie et un bus de communication.

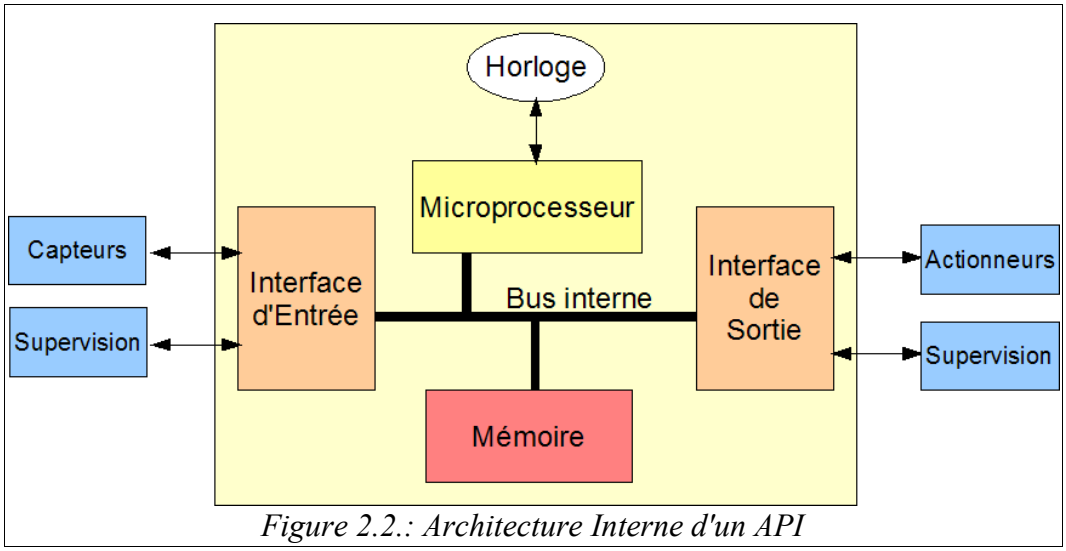


Figure 2.2.: Architecture Interne d'un API

Pour situer les **APIs** au sein des autres plates-formes d'exécution, on peut reprendre la classification décrite par [Harel 1985] et considérer un **API** comme un *système réactif* réagissant aux évolutions de son environnement à une fréquence bien déterminée, contrairement aux *systèmes transformationnels* dont les entrées ne sont pas modifiées durant l'exécution et les sorties seulement produites à la fin (pas de cycles répétitifs d'acquisition et d'écriture), et aux *systèmes interactifs* qui réagissent de façon anarchique aux stimulations extérieures (les **OS** (*Operating System, Systèmes d'Exploitation*) traditionnels).

On peut alors décrire les caractéristiques principales d'un **API** [Jimenez 2001] :

- **Déterminisme temporel** : sur le temps de réaction du système à une stimulation extérieure. On parle alors de temps réel, c'est à dire qu'on est certain que le matériel va réagir dans un laps de temps précisé et maximal. Cette contrainte fait partie de la spécification, elle doit être prise en compte dans la phase de conception, et elle doit être vérifiée par l'implémentation. Le respect de cette contrainte requiert une implémentation efficace, mais aussi des outils pour le calcul précis du temps d'exécution (calcul des performances). Ainsi pourra-t-on, par exemple dans le cas des **APIs**, déterminer le temps de scrutation.

- **Sûreté** : quant à la réalisation des tâches automates car le système est

responsable d'environnements où interviennent des personnes physiques et du matériel coûteux.

– **Concurrence des tâches** : permettant de gérer des systèmes plus complexes nécessitant une collaboration entre plusieurs **APIs**. Les cycles ne sont pas interdépendants et doivent pouvoir s'exécuter en parallèle.

– **Déterminisme logique** : pour un état donné des entrées et des variables internes, les sorties résultantes seront toujours les mêmes.

Le microprocesseur est un processeur classique réalisant toutes les opérations logiques contenues dans le programme. La mémoire se divise en deux parties, d'une part la mémoire volatile de type « *Random Access Memory* » (**RAM**) où se trouvent les variables d'entrée et de sortie, et d'autre part, la mémoire morte de type « Read-Only Memory » (**ROM**), « *Erasable Programmable ROM* » (**EPROM**) ou « *Electronically EPROM* » (**EEPROM**) qui est généralement chargée de stocker le programme compilé. La conception de l'**API** est différente d'un ordinateur personnel car les circuits se chargeant des entrées et des sorties sont minimisés et optimisés de façon à accélérer les échanges avec le processeur. Il faut également, en cas de blocage ou de défaut, que les données concernant l'installation, comme l'état de certaines variables critiques, soient conservées. Une redondance des informations est donc maintenue de façon continue pour faire face à des pertes de données lors de défaillances du programme principal. Toutes ces particularités différencient donc un **API** d'un ordinateur classique et font que l'**API** est le seul élément capable de garantir un fonctionnement sûr et durable dans des environnements très contraignants.

2.2.2 Fonctionnement

Les tâches réalisées par un **API** sont effectuées de façon cyclique. Ainsi, dans une première phase de ce cycle, les entrées sont échantillonnées puis stockées dans la mémoire volatile de l'automate et enfin verrouillées pour la durée du cycle. Le programme est alors exécuté et, une fois terminé (ou bien lorsque le laps de temps qui lui est accordé est achevé selon le mode de fonctionnement choisi pour l'automate), les sorties sont actualisées puis verrouillées jusqu'à leur prochaine actualisation. La Figure

2.3 reprend ces phases de fonctionnement.

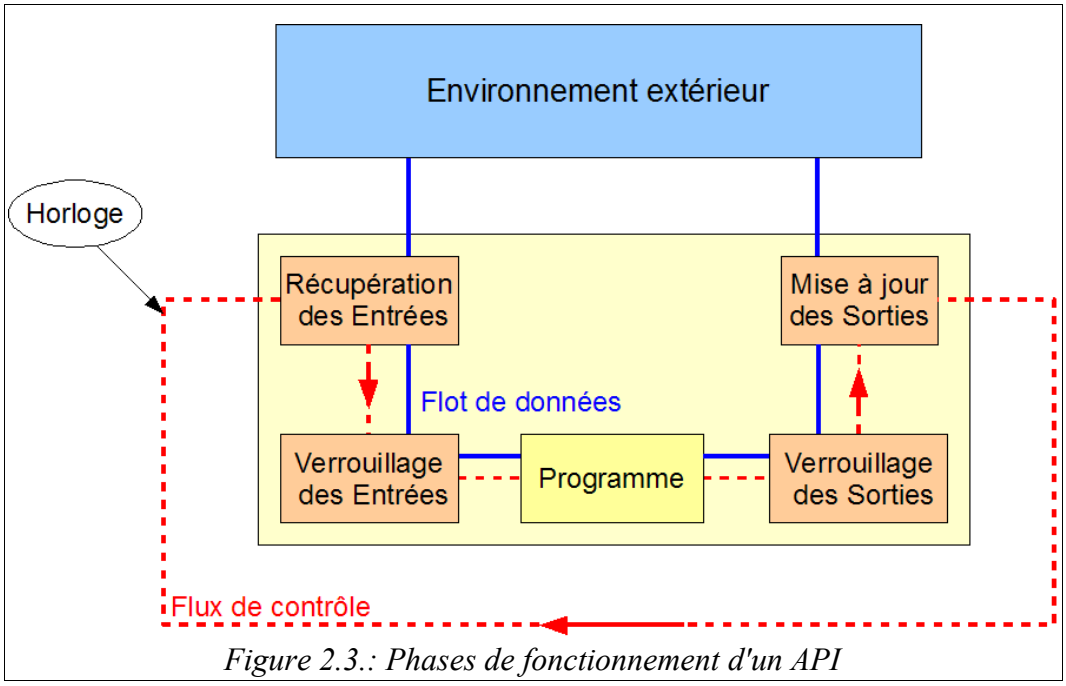


Figure 2.3.: Phases de fonctionnement d'un API

On retrouve en fait la logique des architectures informatiques mises en oeuvre il y a une trentaine d'années [Maler 1999] avec un fonctionnement simple qui permet de contrôler le temps d'exécution de façon déterministe.

Une autre caractéristique essentielle des API et qui est également un élément les distinguant des ordinateurs classiques est la gestion de la mémoire. En effet, l'espace mémoire stocké ou qui va être utilisé (mémoire RAM) est connu à l'avance et borné. Il n'y a pas d'allocation dynamique de mémoire durant l'exécution, ce qui limite d'un côté les possibilités de gestion au niveau de la programmation mais assure d'un autre côté que le fonctionnement ne rencontrera pas de défaillances provoquées par une saturation de mémoire.

2.2.3 La programmation des API

Du fait de leur mode de fonctionnement cyclique et déterministe, les programmes compilés pour être déployés sur les API sont réalisés dans cette optique particulière et les langages de programmation usuels sont très différents des langages que l'on retrouve dans l'informatique. Ceci est en particulier dû, à l'expérience et aux domaines de compétences des personnes qui utilisent et font évoluer ces matériels; en effet elles proviennent notamment du monde de l'électronique ce qui inclut une influence et des sensibilités différentes de celles des informaticiens.

2.2.3.1 La normalisation

Comme pour l'informatique, des efforts de normalisation ont été effectués; plusieurs normes ont été adoptées pour diriger la conception et la programmation des automatismes en particulier:

- La **norme 61131** adoptée en 1993 par l'IEC [IEC 61131 1993], consortium spécialisé dans les domaines de l'électronique et travaillant de plus en plus conjointement avec l'International Standards Organization (ISO). Cette norme est la plus connue et la plus usitée par les automaticiens et elle continue d'évoluer notamment sous l'impulsion de comités techniques tels que PLCOpen et de grandes sociétés spécialistes de l'automatisme (Siemens, Rockwell, Schneider,..). Nous nous intéresserons plus en détails à cette norme dans le chapitre suivant car notre travail de recherche comporte une partie de déploiement des spécifications vers les langages qui y sont décrits.

- La **norme 60848** de l'IEC [IEC 60848 1988] davantage centrée sur la spécification graphique des séquences dans la programmation s'est inspirée d'un langage mis au point par l'Association Française de Cybernétique Économique et Technique (AFCET), s'appuyant sur un déroulement d'étapes et de transitions entre elles. Ce langage, très utilisé en France, a pris le nom de GRAphe Fonctionnel de Commande Étapes/Transitions (GRAFCET). Il a ensuite été intégré dans la norme 60848 sous la dénomination de Sequential Functions Charts (SFC) avec quelques variations, cependant, car plus orienté vers la programmation et moins vers la

spécification.

– La **norme 61499** de l'IEC créée en 2000 puis révisée en 2005 [IEC 61499 2005], reprend le concept de Bloc Fonction (*Functional Bloc* **FB**) de son homologue **IEC 61131** en l'enrichissant d'une gestion des flux d'évènements, l'*Execution Control Chart* (**ECC**). Bien qu'elle soit destinée à remplacer dans l'avenir la norme IEC 61131, on remarque cependant que les grands acteurs de l'automatisme, tout en s'y « *intéressant* » de près n'envisagent pas pour le moment d'intégrer ce mode de programmation dans leurs **IDE** (*Integration Development Environment*, environnement de programmation intégré). La société *Rockwell* possède un prototype d'outil appelé **FBDK** (*Function Bloc Developer Kit*) et la société *ICS Triplex* a, quant à elle, sorti un véritable outil conforme à la norme, **IsaGRAF 5.0**, mais les utilisateurs sont encore très « *frileux* ». De plus, la majorité des automaticiens sont formés à utiliser des méthodes classiques (analyse fonctionnelle, **GEMMA** (*Guide d'Étude des Modes de Marche et d'Arrêt*), logique séquentielle **GRAFCET** (*GRAphe Fonctionnel de Commande Étapes/Transitions*)), ils ne sont pas formés à la logique de réflexion adaptée pour le développement de tels blocs, et se montrent plutôt réticents pour l'apprentissage d'un nouveau mode de programmation avec en plus une méthodologie de conception différente.

Chez eux persiste encore beaucoup de scepticisme quant à l'utilisation de cette norme, et on retrouve plutôt le couplage traditionnel de langages **IEC 61131** qu'ils maîtrisent parfaitement pour mixer des logiques séquentielles et algorithmiques (**SFC** et langage structuré par exemple). Cependant, à n'en pas douter, cette norme s'intègrera au fur et à mesure dans les mentalités car les améliorations qu'elle apporte sont multiples, notamment pour la réalisation d'architectures avec des automates distribués, ensembles que l'on retrouve de plus en plus dans l'industrie aujourd'hui.

Nous verrons dans notre étude que le travail effectué sur la norme **IEC 61131** reste tout à fait valable si l'on devait envisager une évolution vers la norme **IEC 61499**, des travaux s'intéressent d'ailleurs déjà à la relation UML 2/IEC 61499 [Panjaitan 2006], [Dubinin 2004], [Hussain 2006].

2.2.3.2 Évolution des environnements de programmation

Les environnements de programmation automate ont également tendance actuellement à ouvrir vers l'extérieur le format de stockage des programmes. Par exemple les plates-formes de développement *ControlBuild 4.1* de *GEENSYS* et *Unity Pro* de *Telemecanique* ont ouvert leur format d'échange de fichier en choisissant le langage XML pour la description des programmes réalisés. De plus on vit l'apparition de kits de développement logiciel SDK (*Software Development Kit*) permettant le développement de composants logiciels pouvant être appelés pendant l'exécution du programme automate.

2.2.4 Évolution des architectures matérielles

Pour les architectures matérielles, on constate une évolution des technologies vers le concept d'*automate-PC* basé sur des plates-formes de type *PC industriel* [Borges 1997]. Elles représentent une alternative intéressante aux cartes d'extension PC réputées trop peu fiables. Les ventilations sont remplacées par des radiateurs, les mémoires sont de type *Compact Flash* (CF), elles sont directement placées sur les *racks* (rails sur lesquels on retrouve les modules d'entrées et de sorties, les relais, les actionneurs complexes comme les variateurs) de l'armoire électrique d'une installation et possèdent des interfaces de communication pour des bus de type industriel. Le temps réel est garanti par des systèmes d'exploitation (OS *operating system*) adaptés garantissant un ordonnancement des tâches déterministes.

La plupart des fournisseurs de matériels d'automatisme proposent leur propre OS (B&R, Beckhoff par exemple) mais la tendance actuelle en 2007 est à l'ouverture vers des niveaux supérieurs du système informationnel de l'entreprise via Internet par exemple. Il est alors utile d'avoir à sa disposition des environnements multimédias plus répandus sans que cette intégration ne nuise aux performances de l'application de contrôle/commande. Les plates-formes sont alors livrées avec des OS tels que Windows XPe (windows XP *embedded*), Windows Ce, Linux RT, VXWorks pour ne citer que les plus répandus.

2.3 **Les architectures des systèmes automatisés**

Il existe une multitude de combinaisons possibles pour définir une architecture d'automatisme, selon le choix du réseau de terrain et du matériel. Nous présentons néanmoins sur la Figure 2.4 une architecture classique que l'on retrouve notamment dans le domaine industriel servant de cadre à cette thèse.

Le niveau supérieur correspond aux réseaux qu'il est possible de mettre en place entre le superviseur et des postes situés ailleurs dans l'entreprise. On voit également qu'il est possible d'avoir un réseau directement connecté à internet.

Le second niveau virtualise un réseau d'échange entre les systèmes de contrôle (par exemple des automates) et le superviseur. Ce réseau a des contraintes temps réel dépendant du niveau de criticité des ordres envoyés à partir du superviseur. Pour cette raison nous avons représenté un réseau Ethernet industriel basé sur des *switchs* et plus robuste qu'un réseau Ethernet classique.

Dans la partie inférieure de l'architecture on retrouve le matériel gérant directement les entrées/sorties du système et pouvant être relié de différentes manières à l'automate maître du réseau, selon la typologie et le protocole choisis.

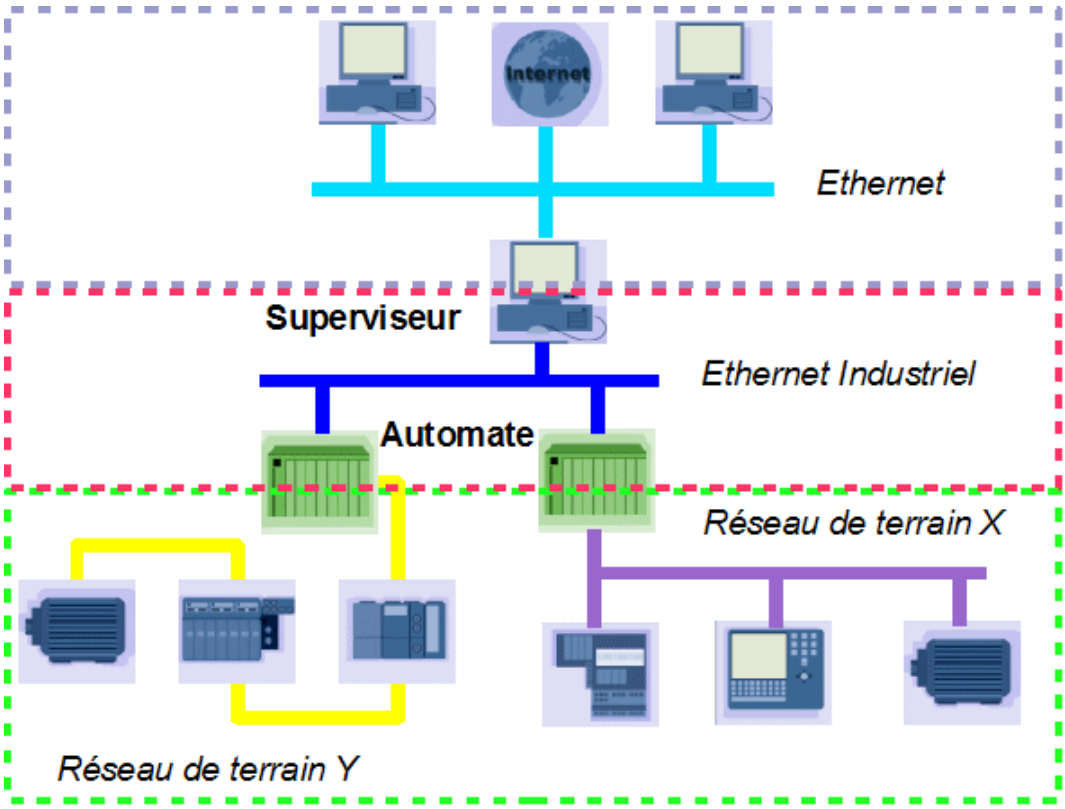


Figure 2.4.: Architecture classique d'automatisme

Si l'on se place dans une perspective de conception, et si l'on se concentre sur un module physique donné, par exemple un convoyeur, on voit que la spécification complète de l'élément devra couvrir ces trois niveaux d'architecture.

En effet, on peut retrouver des besoins au niveau d'internet, si l'état du convoyeur dit pouvoir être consulté sur ce médium de communication à partir d'ordinateurs distants, des besoins en terme d'interface de dialogue pour l'opérateur, afin de lui permettre de faire fonctionner le convoyeur en mode manuel par exemple. Nous aurons ensuite des spécifications au niveau du programme de l'automate pour le contrôle en mode continu. Enfin, il faudra prendre en compte des considérations de câblage électrique pour l'introduction du convoyeur dans le système global, ainsi que des problématiques mécaniques pour le positionnement dans l'implantation physique et l'interaction du convoyeur avec les autres éléments existants.

2.4 Conclusion

Ce chapitre nous a permis d'introduire les spécificités principales des automates par rapport à d'autres plates-formes d'exécution notamment en termes de déterminisme temporel et de robustesse. Ces particularités font que leur mode d'implémentation est régi par des outils, des standards et des langages spécifiques à l'automatisme.

C'est en grande partie pour cette raison que les méthodes populaires de conception existant dans d'autres domaines n'ont pas trouvé d'écho dans le monde de l'automatisme, n'étant pas adaptées à ses particularités. Des langages de modélisation automates, c'est à dire basés sur une représentation graphique de la logique du programme ont été proposés par l'**IEC**. Ils font l'objet du chapitre suivant et en particulier ceux décrits dans la partie 3 du standard IEC 61131. Nous verrons les logiques particulières de ces modes de programmation afin de préparer la présentation du travail de génération de code illustré dans le Chapitre 6.

CHAPITRE 3 :

LES LANGAGES DE PROGRAMMATION DE LA NORME IEC 61131

3.1 Introduction

La norme **IEC 61131** est constituée de huit parties destinées à définir la structure matérielle des systèmes automatisés et les langages qui permettent de réaliser leur programmation :

- 61131-1 : Aperçu général et Définitions
- 61131-2 : Structures matérielles
- 61131-3 : Langages de programmation
- 61131-4 : Guide utilisateur

- 61131-5 : Spécification du service de Messagerie
- 61131-6 : Communication et Réseaux de Terrain
- 61131-7 : Contrôle Flou
- 61131-8 : Guide pour l'application et l'utilisation des langages de programmation.

Nous allons nous intéresser surtout à la partie trois qui détaille précisément les langages de programmation. On retrouve deux notions importantes autour desquelles se présente la spécification:

- Les éléments communs à tous les langages, qui vont représenter les ressources d'un programme. Ces éléments sont appelés « *modules* » ou plus souvent « *unités d'organisation de programme* » (*Program Organization Unit* **POU**).
- Les langages de programmation

3.2 **Les Éléments Communs**

3.2.1 Les types de données

Ce sont d'abord les types de données que l'on va pouvoir définir dans un programme de façon à imposer un typage strict des variables. Les types de base sont les booléens, les nombres entiers, réels, octets et mots, mais également les dates, heures et chaînes de caractères.

Il est ensuite possible pour un utilisateur de dériver ces types pour se constituer ses propres *types dérivés*, par exemple un canal d'entrée analogique, ce qui permet par la suite d'utiliser cet élément comme un type de données prédéfini dans son programme.

3.2.2 Les variables

Les variables sont des instances de types de données et vont donc représenter les ressources de bases pour l'exécution de la logique du programme. Certaines variables particulières (d'Entrée ou de Sortie ou bien d'Entrée/Sortie) vont être reliées à des entrées et des sorties physiques dans le programme. On leur donne une adresse explicite liée au matériel et ces variables d'Entrée/Sortie (E/S) vont en quelque sorte virtualiser les états réels du système au sein du programme.

3.2.3 Les unités d'organisation de programme

Sont classés dans cette catégorie les sections de programme, les fonctions et les blocs fonctions. Ces unités que l'on retrouve dans les documentations sous l'acronyme **POU** (*Program Organization Units*) sont en fait des abstractions de fonctionnalités que l'on veut pouvoir utiliser plusieurs fois de façon pratique. Ces éléments possèdent des interfaces avec un ensemble de données d'E/S, et un corps contenant une portion de programme réalisé dans l'un des langages de la norme.

3.2.3.1 Les fonctions

Les fonctions permettent d'effectuer des opérations simples ne nécessitant pas la définition de variables internes intermédiaires. Ainsi pour un même ensemble d'entrées, la valeur de retour de la fonction sera toujours identique. On retrouve dans cet ensemble les fonctions standards mathématiques par exemple (addition ADD, valeur absolue ABS, racine carrée SQRT, ...).

3.2.3.2 Les Blocs Fonction

Contrairement aux fonctions, les blocs fonctions (Function Blocks **FB**) disposent d'un état interne persistant entre chaque cycle d'exécution. Ils permettent donc d'encapsuler des logiques plus complexes au sein du programme. Ces éléments sont centraux dans nos travaux de recherche puisqu'ils représentent le lien entre notre approche de modélisation objet à l'aide **UML** et la génération d'objets d'automatismes

directement déployables sur un **API**.

En plus de variables d'E/S situées sur leurs interfaces externes de données, les blocs disposent d'un ensemble de variables internes accessibles ou non par le programme parent selon les droits d'accès imposés. On retrouve sur la Figure 3.1 une vue schématique d'un FB avec l'encapsulation dans une boîte noire d'un comportement interne programmé dans l'un des langages de la norme.

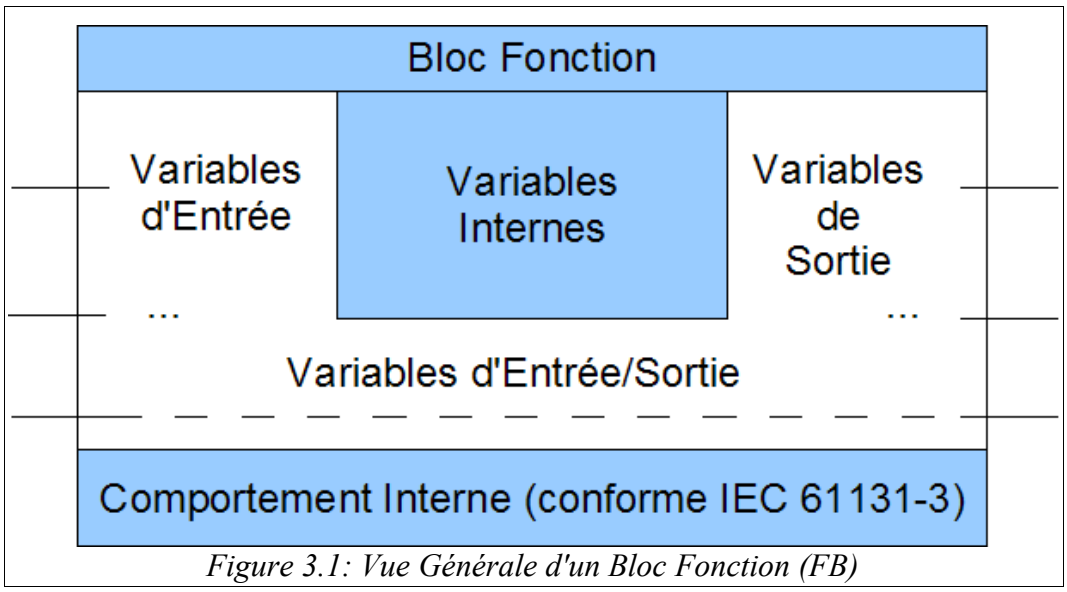


Figure 3.1: Vue Générale d'un Bloc Fonction (FB)

Les FB sont très utiles car ils permettent l'abstraction de logiques complexes et récurrentes au sein d'un programme, augmentant l'efficacité et la rapidité de mise en oeuvre de fonctionnalités présentes à plusieurs emplacements dans le programme, par exemple une boucle de régulation de température, un régulateur Proportionnel Intégral Dérivé (PID). Outre le gain de mémoire obtenu en multipliant des instances de variables au lieu de retrouver la même portion de code plusieurs fois, ils permettent également de protéger des parties de code automate en les rendant inaccessibles à un utilisateur non autorisé à modifier le bloc ce qui est utile lorsque, comme dans la société Newtec, les machines sont fournies avec un code accessible par le service maintenance d'une entreprise cliente. Nous verrons plus en détail les différentes utilisations de ce bloc

lorsque nous décrirons le déploiement des spécifications avec l'extension SysML.

3.2.3.3 Les Programmes

Les programmes représentent le niveau d'abstraction le plus élevé dans la programmation puisqu'ils encapsulent des fonctions, des blocs fonctions et des expressions décrites dans l'un des langages de la norme. On les retrouve souvent dans les IDE du marché décomposés en « *sections* ». Une section est toujours associée à un langage donné et possède un emplacement dans la séquence d'exécution du programme global.

3.2.3.4 La structure des diagrammes SFC

Contrairement aux autres langages de la norme, la représentation **SFC** n'est pas réellement considérée comme un langage mais plus comme un moyen de structurer l'exécution et la séquence de différents sous-programmes. Ainsi dans un diagramme SFC on retrouvera différents langages utilisés indépendamment les uns des autres.

Comme précisé précédemment, le **SFC** s'est inspiré du **GRAFCET** mais également de la logique des réseaux de Pétri. On retrouve donc un enchaînement d'entités qui pourront être des « ***Étapes*** » ou des « ***Transitions*** **».** A chaque étape sont associées une ou plusieurs « ***Actions*** » qui sont déclenchées successivement lorsque l'étape correspondante devient active. Pour passer d'une étape à une autre il faut que le programme lié à la transition effectue la mise à 1 de l'état booléen de la transition, afin de permettre le passage du « *jeton de contrôle* » à l'étape suivante.

On retrouve sur la Figure 3.2 un exemple de diagramme **SFC** avec au niveau de la Transition 1, une section de programme décrite en langage **LADDER** (LD) et en langage structuré pour l'Étape 1. On peut ainsi décomposer un problème de commande en éléments de programme individuels et maintenir une vision de l'ensemble cohérente et lisible.

Comme précisé dans [Roussel 1999], le SFC s'inspire fortement de la norme **IEC 60848** et n'est pas un langage **IEC 61131-3** à proprement parler. Les deux normes sont en fait complémentaires et permettent de représenter la structure d'un programme

puis de détailler sa logique interne.

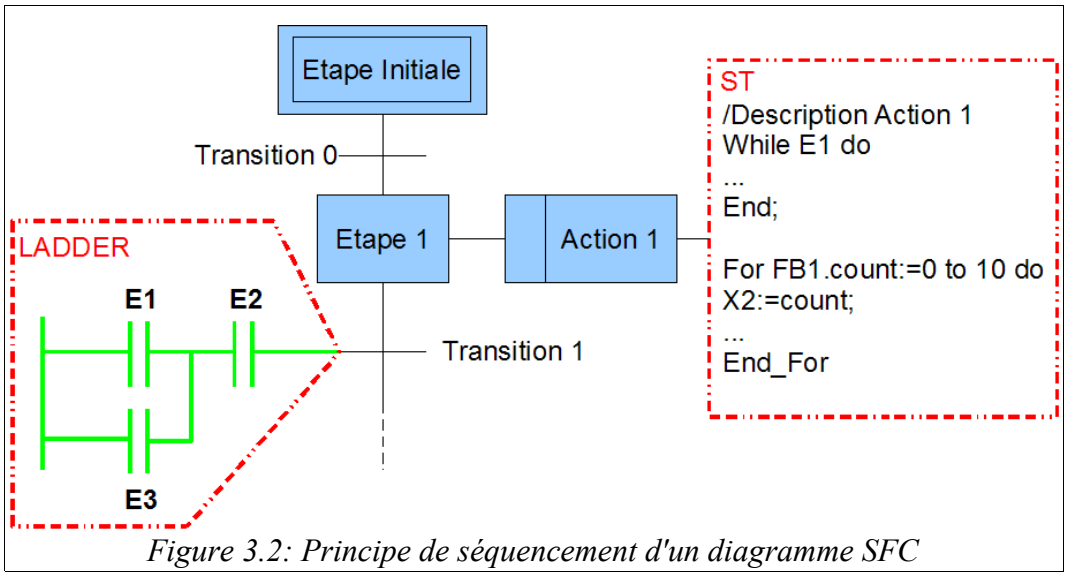


Figure 3.2: Principe de séquencement d'un diagramme SFC

3.3 **Les langages de programmation IEC 61131-3**

Comme énoncé précédemment, la norme **IEC 61131-3** décrit quatre langages de programmation dont les représentations sont radicalement différentes au niveau de la syntaxe. On distingue deux langages textuels (texte structuré et liste d'instructions) et deux langages graphiques (ladder et diagramme de blocs fonction). Chaque langage est né d'une influence particulière d'un domaine (électronique, informatique, process) et plusieurs critères interviennent dans le choix du langage lors d'une nouvelle implémentation:

- Les affinités du programmeur avec un langage
- Le type de problème à résoudre
- Le niveau de description du problème
- La structure du système de commande
- L'interface avec les autres programmeurs ou partenaires

Nous présentons dans la partie suivante ces quatre langages de façon succincte.

3.3.1 Le langage Ladder (Ladder Diagram LD)

Appelé également « *langage à relais* », ce langage est né de la représentation des réseaux de contacts et du besoin, en terme de langage de spécification, des personnes réalisant le matériel automate à base de relais électromécaniques. C'est aux États-Unis que ce langage a acquis sa notoriété depuis sa création en 1969 (le plus ancien des langages de la norme est toujours très utilisé outre-Atlantique [Su 1997]). Il permet de visualiser le fonctionnement de la logique programmée sous une forme très proche de la logique câblée ce qui était plus facile à spécifier pour les personnes mettant en place l'installation lorsque les IDE étaient moins élaborés qu'aujourd'hui.

Comme on peut le voir sur la Figure 3.3 on retrouve donc à gauche une ligne verticale représentant la barre d'alimentation, celle de droite étant le côté neutre. Ensuite, on trace des lignes horizontales de connexion allant jusqu'aux bobines représentant les variables que l'on veut modifier, en passant à travers différents types de contacts permettant de tester si les conditions sont remplies (Zone de test) pour que l'on puisse agir sur les bobines ou lancer des opérations particulières (Zone d'action).

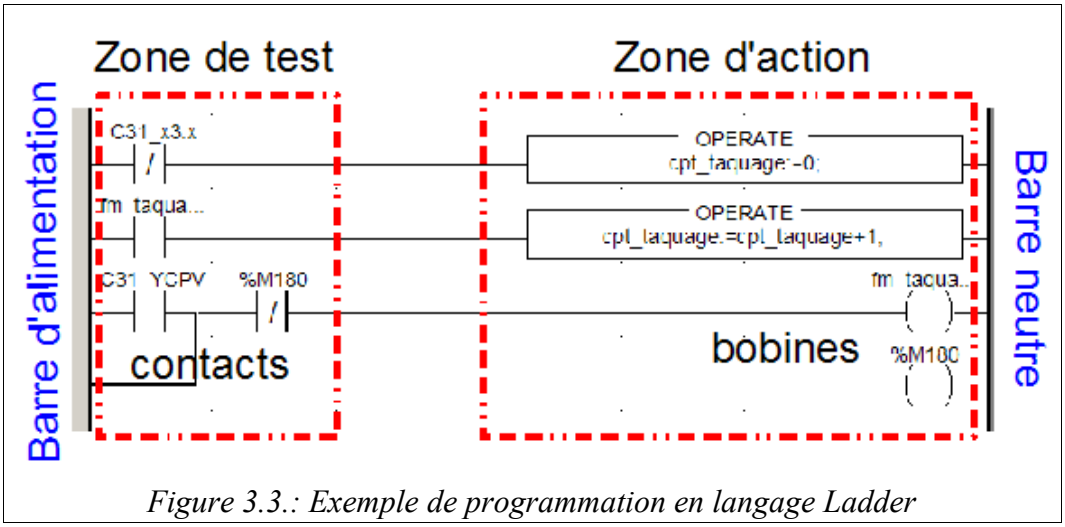


Figure 3.3.: Exemple de programmation en langage Ladder

Un des inconvénients majeurs de ce type de programmation est l'obligation de n'avoir qu'une seule bobine correspondant à un type de variable. Lorsqu'on veut ajouter des conditions de passage, il faut donc généralement revenir en arrière et modifier

la zone de test ce qui peut être fastidieux.

3.3.2 Le langage Texte Structuré (ST)

C'est le langage avec le plus haut niveau d'abstraction de la logique matérielle au niveau de la norme. En effet, influencé par des langages informatiques tels que **ADA**, le **Pascal** et le **C**, il contient toutes les caractéristiques d'un langage de programmation moderne avec la possibilité de définir facilement des boucles « *while* » et « *for* », des blocs conditionnels « *if* » et « *case* » permettant au programmeur de résoudre facilement des logiques de contrôle complexes au niveau de l'écriture du code. Dans le domaine industriel on le retrouve très fréquemment utilisé pour décrire le comportement interne des blocs fonctions non triviaux.

```
(*CONTROLE FLUX SUR A1*)
(*pas accumulation sur A2 et intro colis en cours==>FIFO*)
if V1_Accu_Marche and not (V1_Accu2 or v_cpt_tran=0) AND GEN_FD_S5 THEN
        A_INT2:=0;
        FOR V1_FLUX_INDEX:=1 TO 29 DO
                A_INT2:=A_INT2+V1_FLUX_REGISTRE[V1_FLUX_INDEX];
                V1_FLUX_REGISTRE[V1_FLUX_INDEX]:=V1_FLUX_REGISTRE[V1_FLUX_INDEX+1];
        END_FOR;
        IF V1_E_A1 THEN
                V1_FLUX_REGISTRE[V1_FLUX_INDEX+1]:=1;
                A_INT2:=A_INT2+1;
        ELSE
                V1_FLUX_REGISTRE[V1_FLUX_INDEX+1]:=0;
        END_IF;
        (*mise a zero partie registre si accumulation*)
        FOR V1_FLUX_INDEX:=31 TO 49 DO
                V1_FLUX_REGISTRE[V1_FLUX_INDEX]:=0;
        END_FOR;
        V1_FLUX_INDEX2:=31;
        V1_FLUX_OK:=(A_INT2>15);
END_IF;
```

Figure 3.4.: Exemple de programmation en Texte Structuré

On voit sur la Figure 3.4 un exemple de logique dont la mise en oeuvre aurait été particulièrement lourde dans un autre langage que le langage **ST**, avec à l'intérieur d'une instruction conditionnelle, la présence de boucles « *For* » et d'autres blocs « *If* ». Le langage ST apporte donc un réel confort de programmation mais s'éloigne beaucoup de la logique du câblage électrique.

Ce langage est donc particulièrement intéressant lorsqu'on cherche à prendre de la distance avec une implémentation matérielle lors de la phase de conception du programme. On focalise les efforts sur un aspect algorithmique de la programmation ce qui permet de la rendre plus lisible et plus cohérente en terme de logique de contrôle. Par contre la performance et la rapidité du code compilé dépendront de la qualité du compilateur et seront généralement moindres que pour des langages tels que le langage Liste d'Instructions.

3.3.3 Le langage Liste d'Instructions (IL)

A l'origine destiné au codage des applications critiques, le langage impératif séquentiel **IL** a été intégré dans la norme pour permettre aux programmeurs de produire des instructions proches de celles qui seront finalement traitées par le microprocesseur. On fait souvent l'analogie avec du code assembleur car la syntaxe est très similaire; il permet d'obtenir des performances supérieures à celles des autres langages ce qui représente en revanche l'inconvénient de donner un code extrêmement difficile à lire et à maintenir. C'est un langage textuel où chaque ligne de programme est composée d'une « *étiquette* » et d'une « *instruction* ». Nous ne nous intéresserons pas plus à ce langage par la suite, le gain de temps d'exécution apporté par ce langage diminuant avec les capacités des APIs actuelles.

3.3.4 Le Diagramme de Blocs Fonction (FBD)

Ce langage graphique est lui aussi très fortement lié au domaine de l'électronique car il hérite de la spécification des circuits électroniques en terme d'assemblage de composants pour la réalisation d'un circuit. De même, à partir de « *composants* » ainsi construits, on peut à nouveau les incorporer dans une structure de niveau supérieur. On retrouve la logique d'encapsulation de l'approche objet en informatique mais la conception est différente par rapport aux autres langages, puisqu'on va d'abord penser en termes de composants avant de penser réellement à la logique du programme qui sera, elle, réalisée en connectant les éléments ainsi définis.

Cette approche est souvent difficile à appréhender pour des automaticiens

qui n'ont jamais utilisé cette approche de programmation. On n'écrit pas des flux de contrôle entre les éléments mais des flux de données, en reliant, entrées et sorties des différents FB.

L'ordre d'exécution est ensuite déterminé par la règle de « *précédence* », en général de bas en haut et de gauche à droite, entre les blocs qui ne s'exécuteront que si toutes leurs entrées ont bien été mises à jour dans le cycle en cours.

3.4 **Le langage IEC 61499**

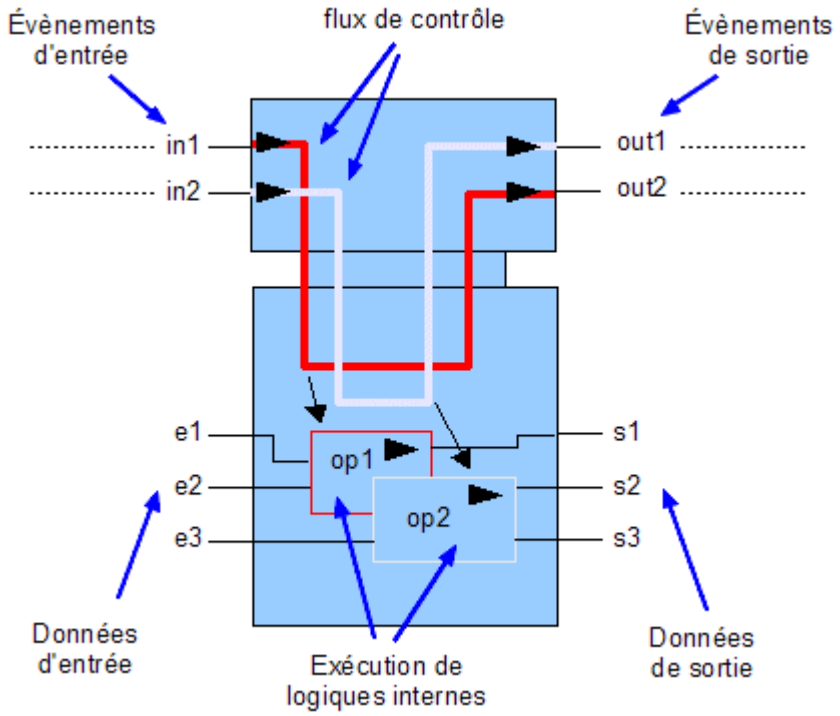


Figure 3.5.: Bloc exécutable dans la norme IEC 61499

Ce langage se consacre entièrement à la définition d'un seul concept structurel qui est le bloc exécutable. Il étend la définition du bloc fonction que l'on connaissait dans la norme **IEC 61131** en lui ajoutant une gestion des évènements qui interviennent en entrée du bloc.

Précédemment lorsque le flux de contrôle arrivait sur le bloc, celui-ci était déclenché et les différentes sections de programmes définies dans le bloc étaient exécutées successivement. Une fois les opérations effectuées, le flux de contrôle repartait vers d'autres connexions. Par contre le bloc exécutable permet une gestion plus fine de ces évènements puisqu'il assigne au bloc une machine à états d'exécution appelée **ECC** (*Execution Control Chart*) combinant l'association des évènements et les sections à exécuter correspondantes. Le bloc n'exécute donc que les opérations désirées.

La Figure 3.5 montre une vue schématique de la représentation graphique des blocs exécutables. La distinction est faite entre la partie haute chargée de traiter les évènements et la partie basse équivalente à la représentation des blocs fonction classiques et affichant sur ses façades les données d'entrée et de sortie.

L'arrivée d'un événement sur une entrée déclenche le déroulement d'un certain nombre d'actions qui vont utiliser puis mettre à jour des données présentes statiquement au niveau du bloc (connexions permanentes).

Comme nous l'avons précisé dans le chapitre précédent, cette notation n'est pas encore utilisée par les automaticiens et nous nous concentrerons pour la suite du rapport sur les blocs fonction classiques. Cependant la démarche proposée reste tout à fait valable dans la perspective d'une adoption ultérieure de cette norme dans la communauté de l'automatisme.

3.5 **Conclusion**

Nous avons passé en revue les langages de programmation de la norme **IEC 61131** permettant une représentation et une implémentation graphique de la logique d'exécution de l'automate. Cependant, la modélisation se situe à un niveau très bas de l'implémentation et peu d'outils sont offerts à l'automaticien pour gérer l'organisation globale de son programme. Cela vient en grande partie du fait qu'il existe des modes de programmation favorables en fonction du type d'automate et de l'environnement de développement utilisé. Pour une même logique de commande, on peut donc avoir des programmes différents au niveau du langage et qui seront tous implémentés intégralement sans la possibilité de passer de l'un à l'autre directement, à l'heure actuelle en tout cas.

Dans la partie suivante nous remontons dans les étapes de conception et passons en revue des méthodes classiques de représentation de système pour identifier si possible des méthodologies existantes de conception de programme qui pourraient convenir au domaine spécifique des systèmes automatisés.

CHAPITRE 4 :

MODÉLISER POUR CONCEVOIR

4.1 La Modélisation

Lors de la définition de systèmes complexes, il est souvent délicat de garder une vision globale et cohérente de la structure de l'ensemble et surtout des interactions internes au système. C'est d'autant plus vrai lorsque le développement s'effectue en parallèle dans plusieurs corps de métiers. La modélisation permet d'améliorer la lisibilité du processus de conception mais également du système lui-même en suivant une logique s'appuyant sur plusieurs principes:

- L'abstraction du système ou de la logique que l'on veut modéliser permet d'identifier les principales fonctions et aspects du système selon la perspective particulière d'un certain observateur. On étudie donc le sujet modélisé en ignorant les détails trop spécifiques ou n'appartenant pas au contexte dans lequel l'abstraction prend du sens [Clark 2004].

– La simplification permet de représenter l'élément modélisé même sans connaître ou maîtriser toutes les lois réelles influençant le système.

Il existe de multiples définitions mais si l'on devait en choisir une pour le terme modèle elle serait dans notre contexte scientifique:

DÉFINITION 4 : MODÈLE

Élément conceptuel, visant à la compréhension et au diagnostic, c'est une vue de l'esprit, analytique ou algorithmique représentant des phénomènes et leurs relations.

Un modèle s'appuie toujours sur un langage ou une technique de représentation offrant à un concepteur un moyen de représenter l'objet de ses investigations. Mais sans guide d'utilisation pour utiliser correctement ces outils dans un contexte donné, la modélisation se révèle souvent confuse et inefficace. On utilise donc des méthodologies qui permettent de guider l'analyse et la spécification selon un processus en général itératif et incrémental. Dans les parties qui vont suivre nous allons surtout nous intéresser aux méthodes qui permettent d'aller à un niveau de spécification proche du matériel; nous ne décrivons pas par exemple des méthodes très orientées sur la définition générale d'un projet (**OSSAD**) ou très orientées pour un domaine donné (**AXIAL** d'*IBM* pour les systèmes d'information).

4.2 Méthodes d'analyse et de conception traditionnelles

Elles ont été élaborées dans des contextes spécifiques; elles sont donc très efficaces lorsqu'elles sont utilisées dans leur environnement originel mais montrent rapidement leurs limites pour d'autres domaines d'utilisation ou alors se trouvent à un niveau d'abstraction trop élevé si bien qu'elles ne permettent pas d'effectuer une transition cohérente vers l'étape de réalisation et de mise en oeuvre technologique. Chaque méthode a son propre langage, et les personnes chargées de la spécification amont d'un projet utilisent en général un éventail de méthodes selon la phase de conception dans laquelle ils se trouvent et selon le contexte technologique associé. Il est donc parfois difficile de s'y

retrouver et de mettre en place des liens cohérents entre les différentes spécifications. Nous présentons ici les méthodes et les langages les plus récurrents dans le milieu industriel et qui continuent d'être utilisés aujourd'hui, notamment dans la définition d'architectures comprenant des systèmes automatisés.

4.2.1 La méthode APTE

Pour les petites et moyennes entreprises ou industries (PME / PMI), il est important de se doter de méthodes rapides, simples et efficaces dans leur mise en oeuvre. Appartenant à la famille des méthode d'Analyse de la Valeur (AV), la méthode APTE (du nom du cabinet de conseil français l'ayant définie) fait partie de ces méthodes et permet d'aboutir rapidement à l'expression d'un premier recueil de spécifications. Classiquement, on identifie des besoins globaux pour le système que l'on veut réaliser à l'aide d'un diagramme simple « *bête à cornes* », ensuite on traduit ces besoins en fonctions devant être fournies par le système. Elles se décomposent en Fonctions Principales (FP), Secondaires (FS) et de Contraintes (FC) et on les représente graphiquement sur un « *diagramme pieuvre* » (Figure 4.1).

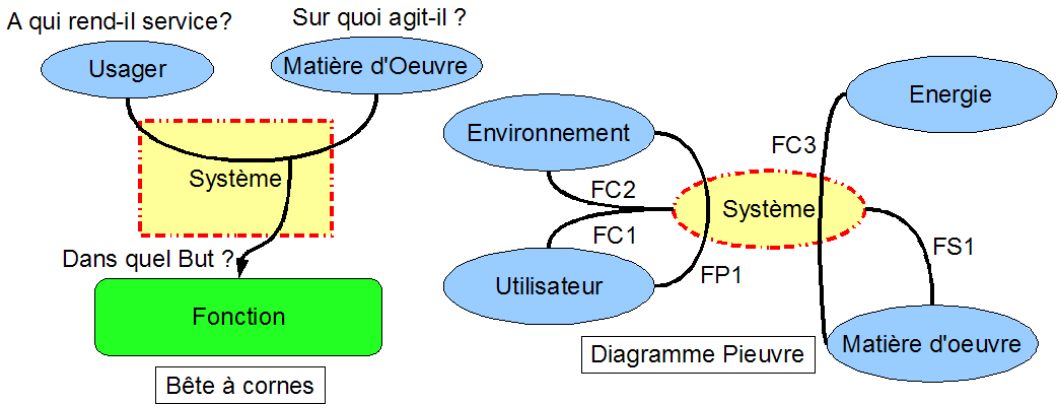


Figure 4.1.: Méthodes graphiques d'analyse préliminaire

L'étape suivante est l'identification de solutions techniques répondant à chacune des fonctions identifiées précédemment. C'est très simpliste et on ne prend pas nécessairement en compte l'interdépendance des solutions ou même des fonctions mais cette méthode permet d'aboutir à un Cahier des Charges Fonctionnel (**CdCF**) rapidement.

On peut alors détailler les solutions retenues à l'aide d'un diagramme **FAST** (*Functional Analysis System Technique*) pour une analyse générale du produit, et un diagramme **SADT** (voir plus loin) pour la description de la structure interne de la solution technique détaillée. La Figure 4.2 reprend les étapes successives à suivre dans ce contexte d'analyse.

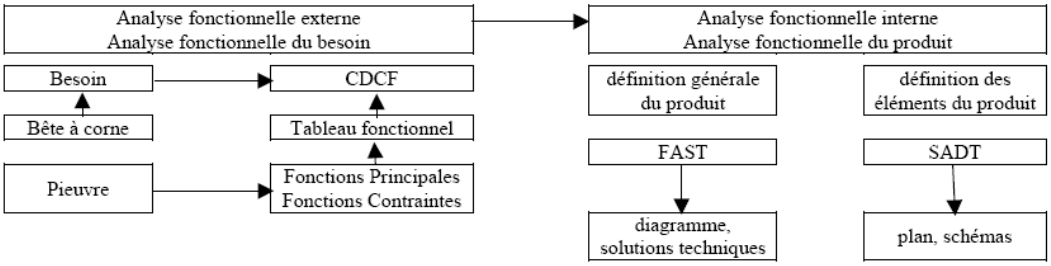


Figure 4.2.: Vue d'un processus d'analyse fonctionnel classique avec les outils associés aux différentes étapes d'analyse

Nous verrons plus tard qu'il est possible de reprendre cette approche en utilisant le langage UML comme support de modélisation ce qui nous permet ainsi d'intégrer cette analyse fonctionnelle au sein d'un modèle plus « consistant ».

4.2.2 Les méthodes basées sur IDEF

Le langage **IDEF** (*Integrated computer aided manufacturing DEFinition language*) a été mis au point en 1993 par l'*US Air Force* [US Air Force 1993]et s'est rapidement popularisé avec notamment son utilisation au sein du Département de la Défense américain (*DoD*). A partir de ce langage, différentes méthodologies ont été construites pour modéliser différents domaines d'application entourant la spécification d'un système: IDEF0 décomposition des activités, IDEF1 flux d'informations, IDEF1x structures de données, IDEF2 simulation, IDEF3 description des processus, IDEF4 conception orientée objet , IDEF5 description d'ontologies, IDEF6 rationalités conceptuelles, IDEF7 méthode d'audit pour les systèmes d'information, IDEF8 interfaces utilisateur, IDEF9 conception dirigée par les scénarii des systèmes d'information, IDEF10 architectures d'implantation, IDEF11 artefacts informationnels, IDEF12 organisation du système, IDEF 13 formalisme tri-schémas, IDEF14 conception de réseaux. Comme on peut le voir, selon le domaine d'utilisation, on a, à notre disposition, une multitude de

méthodes suivant le domaine dans lequel on se trouve. En industrie, on simplifie souvent cette vision normalement constituée de 13 facettes et on se sert généralement des méthodes IDEF0, IDEF1 et IDEF3.

4.2.2.1 IDEF0

Cette méthode s'inspire fortement du très répandu diagramme **SADT** (*Structured Analysis and Design Technique*) mis au point dans la société ***Softech*** [Ross 1977a] [Ross 1977b]et qui dans les années 1980 s'est diffusé dans les services méthodes d'un nombre important d'entreprises européennes comme l'une des méthodes les plus efficaces d'analyse descendante, permettant de décrire un système de façon systémique et hiérarchique comme sur la Figure 4.4. Les briques représentent des fonctions elles-mêmes images d'activités, de processus ou de transformations à l'intérieur du système. Une boîte fonctionnelle appelée *Activité* est reliée aux autres boites et à l'environnement extérieur par l'intermédiaire de flèches comme on peut le voir sur la Figure 4.3, puis peut être décomposée à son tour à un niveau inférieur. La popularité de cette représentation vient du fait qu'elle constitue une approche cohérente et non-ambiguë de décomposition d'un système permettant une communication claire entre les analystes, les concepteurs et les utilisateurs du système.

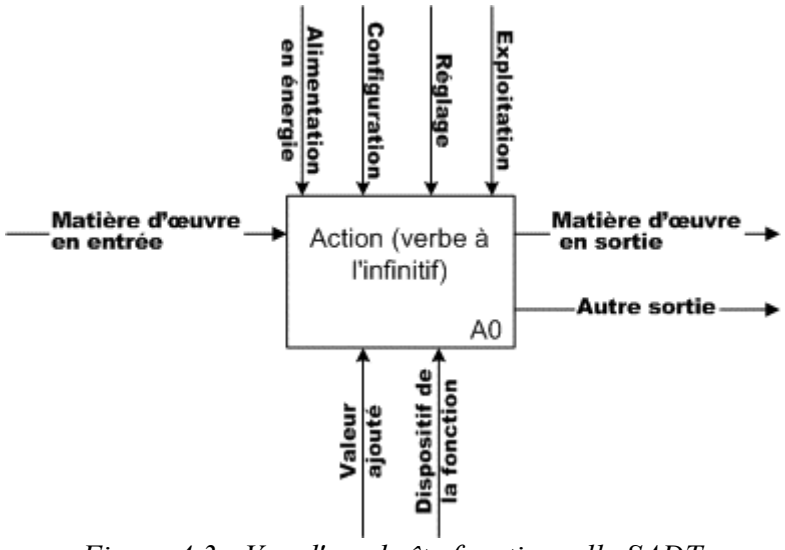


Figure 4.3.: Vue d'une boîte fonctionnelle SADT

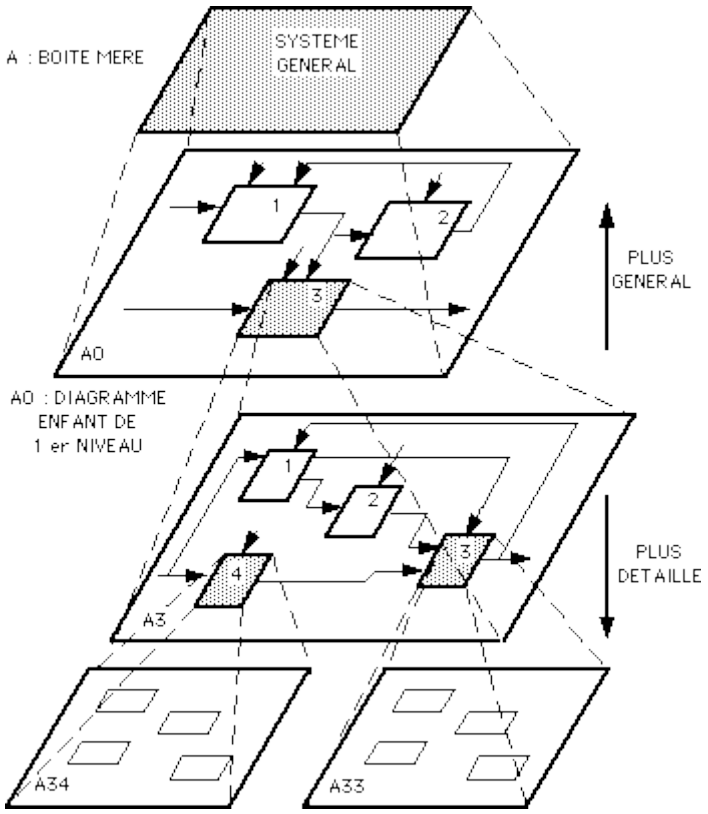


Figure 4.4.: Analyse descendante systémique avec IDEF0/SADT

A l'inverse, on lui reproche l'absence de représentation du processus dynamique lié à une activité en dehors des flux de matière; l'aspect structurel est parfaitement couvert mais les logiques séquentielles, les opérations logiques ne sont pas intégrables sur la représentation. Elle reste cependant la méthode IDEF la plus utilisée en grande partie grâce à sa simplicité de mise en oeuvre. Nous ne présenterons pas les méthodes IDEF1 (basée sur la représentation des flux d'information) utilisée à un niveau supérieur dans la modélisation d'entreprise, et IDEF3 qui se révèle être un modèle des processus trop qualitatif [Galland 2001]et pas aussi performant que les réseaux de Pétri par exemple.

4.2.3 La méthode MERISE

Issue de l'informatique, la méthode **MERISE** (*Méthode d'Étude et de Réalisation Informatique pour les Systèmes d'Entreprise*), développée dans le cadre d'un appel à projets du gouvernement en 1978, prend sa forme définitive sept ans plus tard [Tardieu 1985]. Cette méthode (en particuler son modèle entité-relation) a été très utilisée avant de voir sa cote décroître au sein des architectes de SI au profit d'UML notamment. Elle permet la définition de Systèmes Informationnels (SI) en décrivant en parallèle les données et les traitements. Une intégration de l'approche objet dans la méthode a également vu le jour avec le projet **MERISE**+ [Espinasse 1994], se proposant de se rapprocher de la méthode orientée objet **HOOD** (*Hierarchical Object Oriented Design*). Dans une première étape d'analyse on utilise les deux méthodes (**MERISE** original et **HOOD**), ensuite, on compare les spécifications résultantes selon des axes et critères jugés pertinents.

4.2.4 La méthode GRAI

Acronyme de *Graphes à Résultats et Activités Inter-reliés* elle a été mise au point dans [Doumeingts 1984] puis retravaillée dans [Marcotte 1995]. Cette méthode permet d'analyser les systèmes de Productique en général et a longtemps été utilisée dans les grands groupes manufacturiers français en amont des spécifications plus techniques. Elle s'appuie sur une description du système en suivant plusieurs approches :

- Le modèle conceptuel de référence, qui décrit le système sous trois angles différents: l'angle gestion de production, l'angle structure du système de production, et l'angle structure décisionnelle du système de production qui aboutit à la définition des centres de décision.

- La grille **GRAI** qui permet de représenter la structure décisionnelle périodique du système de pilotage avec la planification de la production des produits, la gestion de ces produits en terme d'approvisionnement des nomenclatures, la gestion des ressources de production mises en oeuvre pour le traitement et le transport des produits (par ressources on entend également-les ressources humaines).

- Les réseaux **GRAI** qui reprennent les centres de décision identifiés dans

la grille et guident l'utilisateur dans l'identification d'activités, dans le sens de processus de transformation, en les classant en deux catégories, activités de décision et activités d'exécution.

De même que pour **MERISE** ou **IDEF**, la méthode **GRAI** présente une utilisation très pertinente lorsque le contexte du projet que l'on veut modéliser n'a pas vocation à évoluer de façon importante dans le futur ce qui est rare à l'heure actuelle comme on l'observe dans toutes les organisations de production.

4.2.5 L'architecture CIMOSA

Nous nous contenterons d'évoquer les modèles **CIMOSA** (*Computer Integrated Manufacturing – Open System Architecture*) composant le cube **CIMOSA** et décrivant les intersections de différents axes de modélisation comme illustré sur la Figure 4.5. Nous n'irons pas plus loin dans la description de cette matrice qui pourtant une des seules méthodologies, avec la matrice de « *Zachman* », permettant de passer en revue tous les aspects de spécification d'un projet.

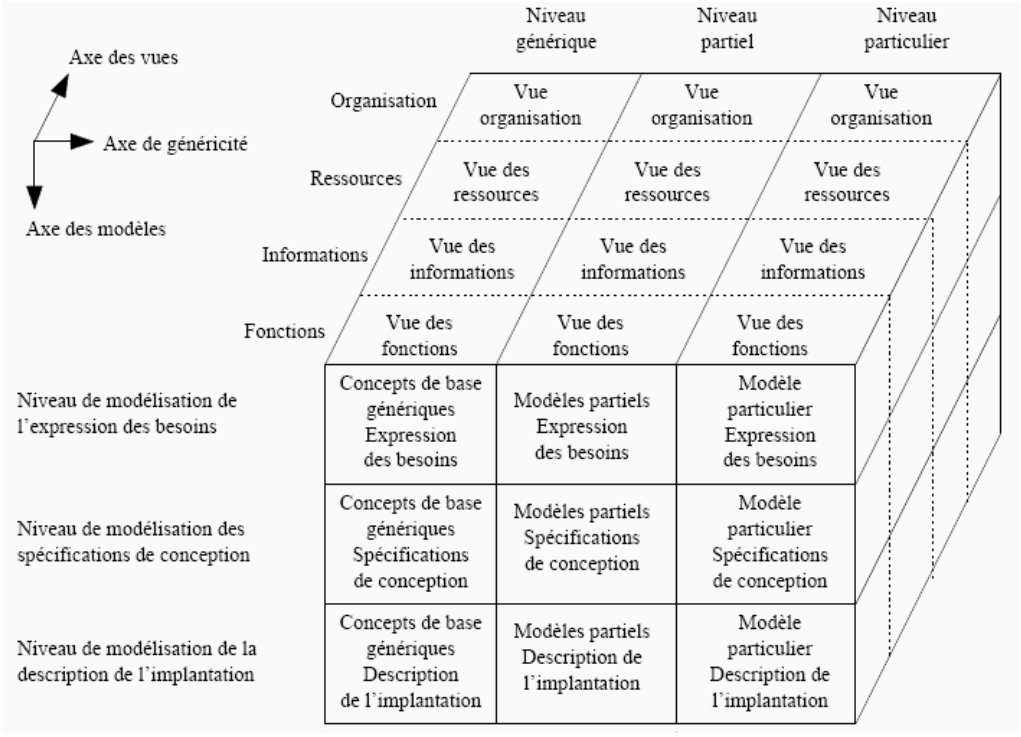


Figure 4.5.: Matrice des modèles composant CIMOSA

Malheureusement, l'absence de guide pour le formalisme de la description de chaque cellule et le grand nombre de cellules de la matrice rend très difficile sa mise en œuvre qui peut vite devenir inexploitable.

4.2.6 Les autres méthodes

D'autres méthodes moins répandues ont également été proposées comme la méthode **PERA** (*Purdue Enterprise Reference Architecture*) de [Williams 1994] et sont venues grossir la population des modélisations fonctionnelles permettant de partir de spécifications de très haut niveau jusqu'à l'implantation physique d'un système.

Ce sont toutes d'excellentes méthodes d'analyse mais elles ne permettent pas de construire une modélisation qui puisse s'adapter aux changements du contexte de réalisation ou de l'évolution des besoins au cours du temps ce à quoi peuvent répondre les méthodes orientées objet [Mertins 1995]; il était néanmoins difficile de ne pas les évoquer dans le contexte de la conception de système de production automatisés pour souligner leurs limites dans ce contexte qui est pourtant le leur.

4.2.7 Les réseaux de Pétri

Les réseaux de Pétri sont une représentation graphique permettant de décrire le comportement dynamique d'environnements logiques sous une forme présentant de nombreuses similitudes avec les diagrammes d'états-transitions de type **GRAFCET**. Le graphe de Pétri se compose de deux noeuds, par lesquels transite le flux de contrôle d'un système, de places et de transitions. Les places, représentées par des cercles, sont actives si un ou plusieurs jetons y sont présents (représentés par des points noirs). Ces jetons peuvent transiter d'une place à une autre (de façon instantanée) si les transitions sont « *sensibilisées* », c'est à dire si les places qui les précèdent contiennent toutes au moins un jeton. Les événements permettant de faire évoluer le graphe correspondent à des ajouts de jetons au niveau de places d'entrées, déclenchant des sensibilisations de transitions et des activations de places en cascade.

Bien que très intuitifs et performants pour la représentation du comportement dynamique d'un système en réaction à des évènements discrets, les réseaux

de Pétri représentent un formalisme particulier ne s'intégrant pas de façon standard dans des langages de représentation structurelle de systèmes.

4.2.8 Les StateCharts

Proposés par Harel [Harel 1987], ces diagrammes d'états-transitions (statecharts) ont fortement influencé le langage **UML** que nous décrivons plus en détail plus loin. Ils permettent de décrire un comportement dynamique sous la forme d'une succession d'états d'un élément et de transitions comportant des conditions de passage.

4.2.9 Le GEMMA et l'AMDEC

Pour les automaticiens, des méthodes moins abstraites et plus axées sur une logique de spécification définissant des modes de fonctionnement, des anomalies et les traitements de ces anomalies sont toujours très utilisées, souvent à la suite des méthodes précédentes. La première et la plus populaire est le **GEMMA** (*Guide d'Étude des Modes de Marche et d'Arrêt*) développé par l'**ADEPA** (*Agence pour le DÉveloppement de la Productique Appliquée*) qui constitue une procédure pour l'analyse d'une logique automate dont on veut déterminer les différents chemins que peut prendre le flux d'exécution et ainsi en prévoir le comportement. L'utilisation du **GEMMA** s'effectue par l'intermédiaire d'un squelette de diagramme graphique générique sur lequel on vient ajouter ou détailler certains éléments et constitue souvent une étape préliminaire à l'élaboration du programme automate en **GRAFCET** car on visualise facilement les séquences et les enchaînements entre les modes de marche.

Le squelette **GEMMA** est illustré sur la Figure 4.6. On peut repérer les grandes catégories du diagramme à savoir :

4.2.9.1 Pour les procédures d'Arrêt.

- A1 (Arrêt dans l'état initial): c'est l'état initial du système. (correspond à l'étape initiale dans la représentation **GRAFCET**), on le représente entouré d'un double cadre.

- A2 (Arrêt demandé en fin de cycle) : Cet état permet de conduire le système à un arrêt à la fin d'un cycle de production. Le système va continuer de produire et s'arrêter lorsque le cycle de production sera terminé. Cet état est utilisé lorsque l'on souhaite alimenter à nouveau un système en matière première.

- A3 (Arrêt demandé dans un état déterminé) : pour conduire le système vers un arrêt différent du précédent. Il permet par exemple de positionner le système dans un état autorisant une intervention sur un système.

- A4 (Arrêt obtenu) : permet de conduire le système vers un arrêt différent de l'arrêt en fin de cycle.

- A5 (Préparation pour remise en route après défaillance) : tâches qui ramènent le système dans un état de remise en route après une défaillance du système. On peut spécifier à ce niveau des interventions opérateur, ou des tâches de contrôle.

- A6 (Mise de la Partie Opérative dans son état initial) : procédures permettant de ramener le système dans son état initial après une remise en route ou une défaillance du système.

- A7 (Mise de la Partie Opérative dans un état déterminé) : permet de stopper le système dans un de transition autre que l'état initial.

4.2.9.2 Pour les procédures de Défaillance.

- D1 (Marche ou arrêt en vue d'assurer la sécurité) : tâches qui permettent de maîtriser le système pendant un arrêt d'urgence. Elles visent à protéger le système et les utilisateurs. Par exemple un cycle de dégagement suivant un arrêt d'urgence appartient à cette catégorie.

- D2 (Diagnostic et/ou traitement de défaillance) : procédures de détection

ou d'action permettant d'identifier et de traiter les défaillances.

– D3 (Production « *tout de même* ») : mode de fonctionnement intermédiaire permettant au système de fonctionner ou d'être mis en route d'une façon non standard, lors de l'exécution de cycles spéciaux ou d'actions manuelles. On parle souvent de modes dégradés ou marches dégradées.

4.2.9.3 Pour les procédures de Fonctionnement.

– F1 (Production Normale) : la machine produit normalement, c'est l'état pour lequel elle a été conçue. C'est à ce titre que le rectangle état a un bord renforcé. On peut souvent faire correspondre à cet état un **GRAFCET** en mode automatique.

– F2 (Marche de Préparation) : cet état est utilisé pour les machines nécessitant une préparation préalable à la production normale, par exemple le préchauffage ou la préparation de l'outillage, le remplissage de la machine, les mises en route diverses.

– F3 (Marche de Clôture) : pour les procédures situées inversement dans le temps à celles décrites dans F2 (cycles de nettoyages, de vidage, fins de série).

– F4 (Marche de vérification dans le désordre) : cet état permet de vérifier certaines fonctions ou certains mouvements sur le système sans respecter l'ordre du cycle (entraîné en automatisme par des forçages par exemple).

– F5 (Marche de vérification dans l'ordre) : dans cet état, le cycle de fonctionnement peut être exploré au rythme voulu par la personne effectuant la vérification, la machine pouvant produire ou ne pas produire.

– F6 (Marche de test) : décrit les fonctionnements périodiques permettant de réaliser certains contrôles, par exemple sur les systèmes de détection de déformation produit, ou de tri, comportant des capteurs qui doivent être réglés, vérifiés ou étalonnés périodiquement.

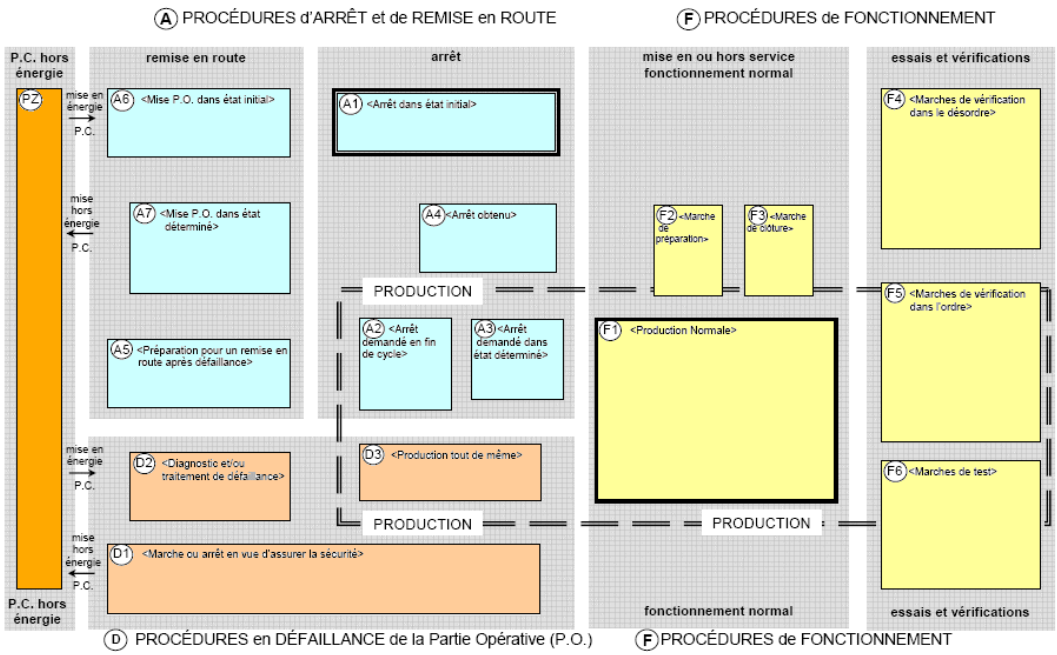


Figure 4.6.: Diagramme GEMMA dans sa représentation vierge

Dans le domaine des systèmes de production, l'évaluation des défaillances a également fait l'objet de travaux pour l'obtention de méthodologies permettant d'analyser et de décrire les différentes anomalies en terme de *Criticité* et la succession des actions correctrices à mener pour relancer le système dans un mode de fonctionnement normal.

La plus populaire est l'**AMDEC** (*Analyse des Modes de Défaillance, de leurs Effets et de leur Criticité*) aussi appelée **FMEA** (*Failure Mode and Effects Analysis*) et développée par l'armée américaine vers la fin des année 40 comme procédure militaire (*MIL-P-1629*) et utilisée ensuite par l'aérospatiale puis l'industrie automobile. Cette méthode permet de décrire graphiquement sur un modèle typique de fonctionnement pour les systèmes de production (cycle automatique, arrêt), les enchaînements d'actions et d'états permettant de revenir à un mode de fonctionnement automatique après une défaillance. On peut ainsi évaluer la fiabilité d'un système en déterminant les effets des défaillances sur les équipements grâce à la détermination de coefficients permettant de déterminer une valeur de criticité pour une défaillance donnée :

$$Criticité\ C = G \times F \times D \quad \text{avec :}$$

G : Gravité

F : Fréquence

D : Détectabilité

Le tableau suivant permet de chiffrer ces facteurs dans le contexte des systèmes de production automatisés :

Valeurs de G	**Critère**
1	Défaillance mineure ne provoquant qu'un arrêt de production faible et aucune dégradation notable (arrêt de production inférieur à 1 heure)
2	Défaillance moyenne nécessitant une remise en état ou une petite réparation et provoquant un arrêt de production de 1 à 8 heures
3	Défaillance critique nécessitant un changement du matériel défectueux et provoquant un arrêt de production de 8 à 48 heures
4	Défaillance très critique nécessitant une grande intervention et provoquant un arrêt de production de 2 à 7 jours
5	Défaillance catastrophique impliquant des problèmes de sécurité et/ou une production non-conforme et provoquant un arrêt de production supérieur à 7 jours
Valeurs de F	**Critère**
	Défaillance inexistante sur matériel similaire (1 arrêt maximum tous les 2 ans)
	Défaillance occasionnelle déjà apparue sur matériel similaire (1 arrêt maximum tous les ans)
	Défaillance occasionnelle posant plus souvent des problèmes (1 arrêt maximum tous les 6 mois)
	Défaillance certaine sur ce type de matériel (1 arrêt maximum par mois)
	Défaillance systématique sur ce type de matériel (1 arrêt maximum par semaine)
Valeurs de D	**Critère**
	Signe avant-coureur de la défaillance que l'opérateur pourra éviter par une action préventive ou alerte automatique d'incident
	Il existe un signe avant-coureur de la défaillance mis il y a un risque que ce signe ne soit pas perçu par l'opérateur
	Le signe avant-coureur de la défaillance n'est pas facilement décelable
	Il n'existe aucun signe avant-coureur de la défaillance

4.3 Conclusion

Il est clair que la modélisation est une condition nécessaire à la réussite d'un projet notamment lors des phases de spécification. Nous avons abordé un ensemble d'outils méthodologiques permettant d'analyser et de guider les concepteurs tout au long du processus de spécification d'un projet de conception d'un système automatisé, et notamment pour le **GEMMA** et l'**AMDEC** d'identifier des éléments très précis pour le contrôle et la commande, mais ils ne s'intègrent pas au sein d'une vraie démarche cohérente de modélisation d'un système dans sa globalité, ce que pourrait permettre **UML**.

Les méthodes et les représentations sont très pratiques et permettent de mettre en œuvre rapidement les premières phases de la spécification. Elle conduisent également à une implémentation rapide d'un système tout à fait fonctionnel dans le cadre d'un développement spécifique.

Les difficultés apparaissent quand le système doit évoluer. On va généralement modifier directement sur le système des éléments pour l'adapter à de nouvelles contraintes ou exigences. On ne repasse pas par les étapes de modélisation, alors qu'il faudrait modifier pour adapter de façon propre et logique le système, mais ce serait un processus beaucoup trop long qui obligerait à retravailler en grande partie voire totalement le modèle fonctionnel initialement réalisé.

Ce manque de flexibilité vient du fait que les modèles font partie intégrante du projet pour lequel ils sont composés. Lorsque le contexte est modifié, il est alors nécessaire de casser des relations dans le modèle et de remettre en cause la structure globale, ce qui représente un coût important en terme de temps de conception.

Dans le chapitre suivant nous introduisons le paradigme objet pour la conception, sensé justement faire face à la variabilité des contextes de projet d'aujourd'hui et offrant la possibilité de composer des bibliothèques de modèles d'objets réutilisables par composition pour de nouveaux projets, et insérables ou détachables pour une application donnée.

CHAPITRE 5 :

L'APPROCHE OBJET

La réutilisation de solutions techniques qui ont déjà fait leurs preuves lors de développements précédents représente une création de valeur indirecte mais réelle pour l'entreprise. De plus, avec l'augmentation régulière de la complexité des systèmes et la difficulté pour les concepteurs de visualiser le développement dans son intégralité, il est devenu essentiel de décomposer les problèmes à résoudre et d'apporter les solutions à chaque élément ainsi identifié. Le concept d'objet est né de ces problématiques dans le métier de l'informatique, proposant de stocker un ensemble de données et d'instructions dans une entité unique qui puisse être ensuite réutilisée à plusieurs reprises dans un contexte donné.

5.1 **Historique**

Dans le domaine logiciel, les technologies de développement ont longtemps reposé sur des langages procéduraux comme le **Fortran** [Backus 1958], le **BASIC** [Kemeny 1964], le **Pascal** [Wirth 1971] ou plus récemment le langage **C** [Kernighan 1983], langages structurés dont la compilation donnait des programmes très rapides à l'exécution. Le premier langage réellement orienté objet est **SIMULA** dans sa version 67 [Dahl 1966] qui a introduit les concepts de classes, de sous-classes, de méthodes

virtuelles, d'allocation dynamique de mémoire repris ensuite dans tous les langages modernes de programmation objet, **SmallTalk-80** tout d'abord [Goldberg 1983], puis le langage **ADA** [Ichbiah 1983], le langage **C++** [Stroustup 1991], ..., avec des évolutions successives comme l'intégration des concepts d'encapsulation et d'héritage.

Ces langages ont rapidement été adoptés par les développeurs, conscients de l'important tournant que l'informatique venait de prendre et des nouvelles puissantes possibilités qui s'ouvraient à eux dans les techniques de programmation, ainsi que du temps qu'ils allaient gagner lors de leurs développements grâce à la réutilisation de fonctions préprogrammées. Cependant, il n'existait pas de méthodologie correspondante à cette époque pour accompagner les développeurs lors des phases d'analyses précédant le travail de codage et pour répondre à la forte demande qui commençait à se faire sentir. Ainsi, dans les années 1990, pas moins d'une cinquantaine de méthodes orientées objet ont vu le jour, se basant sur les différents concepts définissant une approche orientée objet pour les programmes informatiques [Blair 1991], [Berard 1993].

Dans le domaine de l'automatisme, les concepts de programmation n'ont pas évolué aussi vite à cause des restrictions de fonctionnements critiques et de la gestion de mémoire lors de la programmation. Bien que le concept de **FB** (Bloc Fonction) ait été introduit dans la norme **IEC 61131-3**, on est encore loin d'avoir la possibilité d'utiliser les avantages de l'approche objet comme on peut le faire en informatique, c'est d'ailleurs ce que soulignent différents travaux de recherche [Flores 2003], [Bonfatti 1995] qui proposent même un nouveau langage de programmation, [Maffezzoni 1999].

5.2 Les composantes principales de l'approche Objet

5.2.1 L'encapsulation

Un objet définit un ensemble d'attributs qui le caractérisent et qui ne sont accessibles que par l'intermédiaire d'un appel à l'objet si celui-ci autorise cet accès. Les données peuvent donc être protégées ou mises à disposition de l'extérieur selon le choix de la personne qui implémente l'objet. On parle alors d'encapsulation des données. Ces

données sont typées et sont des instances d'autres objets, on peut ainsi réutiliser un objet dans sa globalité pour en caractériser un autre, ce qui permet de structurer beaucoup plus proprement le programme.

De plus, des méthodes sont également définies au niveau d'un objet. Elles permettent d'exécuter des instructions propres à l'objet et peuvent fournir des données vers l'extérieur. Un avantage est que si l'implémentation de la méthode est modifiée et que les paramètres d'entrée et de retour restent du même type, la modification est invisible pour l'élément qui sollicite la méthode, et toutes les instances de l'objet sont également mises à jour.

Au niveau des Blocs Fonction de la norme **IEC 61131-3**, on retrouve partiellement ce concept d'encapsulation avec la gestion de leur propre espace de mémoire et structure de données mais sans la prise en charge des méthodes. Un bloc fonction peut mettre à jour des variables publiques mises à disposition d'autres blocs ou des sorties du système mais il n'est pas possible d'avoir une sollicitation par un bloc client pour obtenir une valeur de retour. On se trouve donc dans un échange de données statique où les correspondances entre variables sont effectuées par un « *câblage* » des données lors de la programmation. Ceci est expliqué en partie par la non allocation dynamique de mémoire, on refuse la création d'une donnée par un bloc fournisseur pour renvoi à un bloc client. Celui-ci doit avoir la donnée créée pour qu'elle puisse être mise à jour, l'intérêt de passer par une méthode est donc limité. On retrouve l'influence forte de l'électronique avec une vue de l'architecture des blocs sous forme d'un circuit avec ses composants et les liaisons électriques positionnées entre leurs différentes pins.

Nous verrons par la suite que c'est cette notion de liaison des données statiques qui fait défaut dans le langage de modélisation **UML 2.0**. Il est impossible de représenter de façon pertinente les connexions statiques entre différents blocs avec l'influence que l'ordre de la séquence bloc-liaison-bloc a sur l'ordre d'exécution des blocs fonction eux-mêmes.

5.2.2 L'héritage

A l'instar de l'être humain, un objet peut hériter des caractéristiques d'un ou

de plusieurs éléments qui sont alors des objets parents. En informatique on va parler de classe mère et de classe fille et retrouver une arborescence ressemblant en quelque sorte à un arbre généalogique où un objet hérite des attributs et des méthodes des différents objets qui sont à un niveau supérieur et sur une branche commune.

L'héritage permet de spécialiser des éléments sans avoir à redéfinir la structure entière de l'élément de base ce qui est très pratique et permet de gagner du temps dans la phase de programmation. Malheureusement, la notion d'héritage est absente de la spécification standard IEC 61131-3 dans sa version actuelle comme le souligne plusieurs travaux sur l'approche objet en automatisme [Bonfe 2005]. C'est l'une des raisons pour lesquelles les automaticiens n'utilisent pas toujours les blocs fonctions dans leur programmation car dés qu'un bloc développé va se retrouver dans un contexte spécifique et que le développeur va vouloir traiter un élément supplémentaire au niveau du bloc, il va devoir reconstruire entièrement un nouveau bloc fonction pour le personnaliser. Il va ainsi perdre de la place mémoire et perdre un temps précieux lors de la conception de son programme.

Pourtant, il est important d'essayer de maintenir la création de blocs fonctions en attendant une évolution de la norme vers un standard qui intégrera l'héritage dans la conception des objets d'automatisme. La récente norme **IEC 61499** ne prend pas non plus en charge cette caractéristique objet mais les apports d'une conception réellement orientée objet dans la structuration et les possibilités de réutilisation qu'elle induit sont importants dans la cohérence d'un programme. Dans nos travaux nous proposons de simplifier ce problème en utilisant la modélisation objet **UML** et son extension **SysML** avec l'idée d'une possible génération de code conforme IEC 61131-3. Nous présentons cette approche dans les chapitres suivants. On peut ainsi par exemple représenter la notion d'héritage au niveau du modèle et ces types de relations sont ensuite interprétés selon la cible de déploiement. Si c'est un langage IEC 61131-3, plusieurs blocs devront être générés. Si la norme évolue, la génération pourra être modifiée de façon à générer des blocs dans lesquels la définition de base commune à plusieurs blocs ne sera pas répétée mais le modèle restera le même que dans le premier cas.

5.2.3 Le polymorphisme

Le polymorphisme est directement lié à l'héritage. Ce concept représente la capacité d'une famille d'objets à proposer une méthode qui a un nom unique mais dont l'implémentation sera différente selon l'objet utilisé, augmentant ainsi la généricité dans la programmation. On sait donc qu'une famille d'objet implémentera la méthode que l'on veut utiliser sans avoir à s'occuper de la façon dont elle réalisera la méthode.

Si on se projette à nouveau dans le domaine de l'automatisme, la possibilité d'utiliser le polymorphisme reviendrait à pouvoir spécifier dans un bloc parent une section de programme avec un nom donné, et à pouvoir redéfinir le contenu de cette section au niveau du bloc fils en gardant le même nom de section et le même emplacement dans l'ordre d'exécution des sections.

Nous avons le même raisonnement que lors de la discussion sur l'héritage, à savoir, nous privilégierons l'aspect objet au niveau modélisation en utilisant le polymorphisme à ce niveau. Ensuite, lors de la génération des blocs fonctions, nous aurons de toute façon des blocs distincts sans relation d'héritage, les sections héritées seront donc recopiées à nouveau et les sections redéfinies seront implémentées comme de nouvelles sections.

5.3 **Les Méthodes Orientées Objet**

5.3.1 Intérêt ?

Comme précisé en introduction, les entreprises doivent faire face à des demandes clientes toujours plus pressantes, toujours plus spécifiques et toujours plus complexes. On cherche alors à aller plus vite, les standards ou les cahiers des charges globaux sont adaptés très rapidement pour un contexte donné ce qui aboutit à des erreurs de spécifications, des points qui ne sont pas approfondis voire occultés, et finalement des défauts sur les systèmes qu'il faut rattraper en investissant du temps et des moyens supplémentaires qui ne sont pas prévus dans un premier temps, mettant en péril la rentabilité des projets.

Pourtant le réflexe paraît logique. Lorsqu'un client commande une fonctionnalité donnée, pris par le temps, on choisit une fonctionnalité proche déjà spécifiée et pour laquelle le processus de développement est connu et on tente de l'adapter en modifiant plus ou moins profondément les parties du développement et les matériels mis en jeu. Les standards ne sont plus des éléments que l'on replace tels quels, mais des bases que l'on remanie sans méthode particulière de façon à ajouter les fonctionnalités souhaitées ou à supprimer celles qui ne sont plus utiles.

On s'aperçoit alors des limites d'une approche se limitant à une analyse descendante (*top-down approach*) comme c'est le cas dans les méthodologies traditionnelles présentées précédemment où l'on part de la spécification des besoins du système global pour identifier les fonctions que doit assurer le système et où l'on détaille ces fonctions dans le contexte particulier du système. Se pose alors la question de la réutilisation. On peut imaginer pouvoir facilement réutiliser des fonctions dans le cadre d'autres spécifications mais la plupart du temps elles sont trop profondément liées à leur utilisation initiale et ne peuvent pas être réutilisées de façon brute. En effet, elles n'ont pas été spécifiées dans un cadre d'abstraction permettant de prendre du recul par rapport au système parent, et de se concentrer purement sur la fonction, ce que prône le concept d'objet, à savoir spécifier des éléments indépendants, comprenant leurs propres logiques et propriétés et offrant des services et des données qui seront toujours identiques quelque soit le contexte d'utilisation.

La spécification d'objets est une approche qui a l'avantage de prendre également en considération une analyse de type montante (*bottom-up*) dans laquelle on va concevoir des fonctionnalités et les encapsuler dans des éléments, les objets, qu'il sera par la suite possible d'intégrer dans des systèmes plus importants par composition. Cette démarche favorise la réutilisation [T.O. Agency 1995]. On peut alors d'une part affiner un système en se focalisant sur le découpage en objets et non plus en fonctions, ou d'autre part, lorsqu'un objet n'est pas disponible, concevoir ce dernier comme un assemblage plus ou moins complexe d'éléments déjà spécifiés et qui sont stockés dans des bibliothèques.

Un autre problème rencontré lors de l'utilisation des méthodologies traditionnelles est la définition des frontières du système étudié. En effet, lors de l'identification des besoins, on se concentre sur un environnement figé pour pouvoir

répertorier toutes les fonctions nécessaires au système puis on se lance dans la spécification détaillée. Cependant, lorsque ces frontières de spécification évoluent, le risque est de devoir remettre en cause tout ou une partie des spécifications. L'avantage de la composition d'objets est que ce type de modification se traduise par l'ajout ou le retrait d'éléments intégrés au système sans modification de la structure globale des autres éléments intervenant dans le système, ni de l'organisation de l'application.

5.3.2 Les prémices

Avant d'adopter le terme de conception orientée objet, on a d'abord parlé de conception structurée dans les années 1960 avec une réelle première mise en forme pour l'informatique dans [Stevens 1974] puis une succession de travaux dans les années qui suivirent [Myers 1975], [Myers 1976], [Myers 1978], [Gomaa 1984], [Ward 1985], [Hatley 1987], [Marca 1988] avec différentes visions de ce que pouvait être une conception structurée. De la même façon, le terme « *orienté objet* » (OOD *Object-Oriented Design*) englobe différents aspects de spécification :

- la définition d'objets individuels et la spécification de méthodes individuelles contenues dans ces objets [Taylor 1990].
- la définition d'objets sous forme d'une structure hiérarchique avec la notion d'héritage et de spécialisation entre les objets d'une même branche.
- La définition d'une bibliothèque d'objets réutilisables [Coggins 1990].
- la définition de l'architecture globale d'un système orienté objet.

Beaucoup de méthodes ont décrit des moyens de couvrir ces aspects mais à chaque fois pour un langage de programmation bien précis [Jalote 1989], [Mullin 1989] ce qui n'a pas donné de méthodologie réellement formelle sur la définition d'une structure composée d'objets; la difficulté étant de rapprocher les analyses formelles et puristes d'une structure, et leurs implémentations réelles au niveau du code dans un langage donné. Les langages gérant de manière différente l'approche objet, on pouvait retrouver pour une même logique de fonctionnement, une organisation et une décomposition du programme complètement différentes selon le langage pour lequel elles étaient réalisées.

Le manque de rigueur et de formalisme des langages pour implémenter

l'approche objet paraît dangereux et se révèle être une source d'erreurs de conception qui peuvent entraîner un manque de flexibilité ou d'efficacité du code. Aussi, les développeurs adeptes d'un langage considèrent-ils ce formalisme, pesant et difficile à réaliser dans la réalité, augmentant le temps de développement du code et alourdissant l'exécution finale du système, contrairement à un programme dédié à une tâche et réalisé d'un seul bloc. La performance a été et est encore un critère en défaveur de l'approche objet puriste dans le monde des programmeurs.

Ainsi, la plupart des langages proposent-ils des compromis et implémentent-ils certains aspects de l'approche objet selon leur évaluation de la pertinence du concept objet à intégrer. Par exemple, certains langages comme le C++ autorisent l'héritage multiple, contrairement à Java qui considère le mécanisme dangereux et lourd. On retrouve également d'autres divergence dans la gestion des exceptions, des « *patrons* » de classe (qu'on retrouve souvent sous les termes anglais *templates*, *parameterized classes*).

Différentes stratégies sont adoptées selon les choix conceptuels et techniques de l'architecte de la structure à étudier ou à concevoir. On peut décrire cependant certaines grandes catégories de méthodologies utilisées pour caractériser les ensembles constitués d'objets.

5.3.3 Méthodologies non-formelles de l'approche objet

Ces méthodologies ne sont pas réellement structurées pour former une marche à suivre précise et infaillible d'analyse et de réalisation pour l'application de l'approche objet. Elles rentrent plutôt dans la catégorie des guides généraux avec des règles à suivre et des principes à respecter pour avancer tout au long de l'analyse et des spécifications. On se concentre alors surtout sur le moyen de définir des objets individuels en dehors du contexte d'un ensemble donné plutôt que sur une démarche de décomposition fonctionnelle de l'ensemble pour aboutir à l'identification de sous-éléments. Dans [Berard 1998] les principales caractéristiques de ces méthodes sont identifiées :

- Comme dit précédemment, on met l'accent sur les objets qui vont constituer l'application plus que sur la compréhension et l'analyse de l'application

globale. On identifie des objets et les fonctionnalités qu'ils proposent ou qu'ils requièrent.

- L'approche suit une analyse montante, c'est à dire qu'on identifie d'abord les plus petits objets en terme de décomposition, puis en remontant on imbrique les objets ainsi définis pour en construire de nouveaux. Il faut souvent tâtonner et l'absence réelle d'objectif final pour l'analyse peut se révéler déroutante, cependant elle permet de constituer une bibliothèque d'objets qui ne sont pas influencés par un projet donné.

Les méthodes basées sur ces caractéristiques ont leur limites quand il s'agit de définir des systèmes critiques très étendus car elles manquent de visibilité sur le processus global d'analyse d'un ensemble. Elles sont cependant rigoureuses pour la définition d'objets individuels et permettent d'identifier efficacement des éléments indépendants et qu'il sera par la suite possible d'utiliser par composition dans l'élaboration d'un système d'un niveau de décomposition supérieur.

5.3.4 Méthodologies dont l'approche objet constitue une partie du cycle d'étude

Dans les années 80, plusieurs études ont proposé l'utilisation d'approches orientées objet en combinaison avec d'autres méthodes d'analyse fonctionnelle plus traditionnelles pour la conception de programmes [Lukman 1991], [Li 1991]. L'inconvénient est de se retrouver avec des incohérences structurelles dans le système dans le sens où des objets peuvent être disloqués pour être localisés dans des fonctions différentes. En effet, on part généralement de l'analyse fonctionnelle en centrant l'analyse sur des fonctions, puis on identifie des objets à l'intérieur de ces fonctions en prenant le risque que ces objets ne soient pas complets car répondant à différentes fonctions à la fois. L'analyse s'éloigne alors de la logique objet et on retrouve les travers de l'analyse fonctionnelle en fixant les spécifications pour un cadre de développement donné; nous ne nous attarderons pas sur ces méthodes hybrides.

5.3.5 Méthodologies formelles objet

Avec la popularité croissante du paradigme objet pour les développements d'applications dans les années 90, une cinquantaine de méthodes orientées objet ont vu le jour durant cette période. Elles ont été en grande partie influencées par le développement des langages de programmation informatiques comme Ada et ont permis d'établir un certain nombre de concepts et d'entités que l'on retrouve dans toutes ces méthodes et qui constituent désormais un vocabulaire bien établi pour discuter des techniques d'implémentation orientées objet. Une *classe* par exemple décrit et centralise les caractéristiques communes d'une famille d'objets ayant les mêmes *attributs* (représentations de l'état et des caractéristiques statiques d'un objet) et les mêmes méthodes (actions dynamiques déclenchables à n'importe quel moment et dont le comportement dépend d'un certain nombre de *paramètres*). Lorsque l'on veut utiliser une classe dans un contexte donné, on va *instancier* cette classe pour obtenir une *instance de classe* ou *objet*. En fonction du contexte d'utilisation, les attributs de ces objets seront modifiés. Nous avons déjà présenté l'*héritage* qui permet d'établir une relation pour montrer qu'une classe appelée *classe fille* ou *sous-classe* hérite des caractéristiques d'une autre classe appelée *classe mère* ou superclasse. Les classes filles ont la possibilité de redéfinir des méthodes ou des attributs: on parle alors de spécialisation; on parlera de généralisation pour exprimer le fait que des caractéristiques communes sont déployées dans des classes filles à partir d'une classe mère. Pour communiquer entre eux, les objets utilisent des *messages* sous forme de noms de méthodes avec des arguments.

L'objectif de la plupart des méthodologies orientées objet est d'aboutir à la spécification de ces classes et on retrouve certaines caractéristiques communes dans l'analyse :

- Un aspect statique permettant de décrire les caractéristiques d'un objet et ses liaisons possibles avec d'autres objets.
- Un aspect comportemental dans lequel on précise le comportement des méthodes.
- Un aspect dynamique pour décrire les interactions entre objets, les états dans lesquels un objet peut se trouver.

Nous présentons succinctement les méthodologies les plus répandues [Baudouin 1996] et qui ont en grande partie influencé la spécification du langage UML (*Unified Modeling Language*) à savoir les méthodes OOA (*Object Oriented Analysis*), HOOD (*Hierarchical Object Oriented Design*), OMT (*Object Modeling Technique*), Booch et OOSE (*Object Oriented System Engineering*).

5.3.5.1 L'analyse OOA (*object Oriented Analysis*)

Cette méthode [Coad 1990] se préoccupe plus de la phase d'analyse d'un système sous un angle de conception objet. Nous allons détailler ses caractéristiques car elle représente l'une des toutes premières réflexions sur une méthodologie d'analyse orientée objet et elle a influencé bon nombre de méthodes ultérieures. Elle s'appuie sur trois principes de base dans l'étude d'une entité cible :

- La différenciation dans la représentation d'un objet complet et ses composants internes.
- La constitution de différentes classes d'objets.
- L'identification des objets élémentaires et de leurs attributs en fonction de l'expérience des personnes qui sont ou seront en relation avec le système.

L'**OOA** suit cinq étapes successives :

5.3.5.1.1 **L'identification des objets**

Cette phase est difficile car elle suit un processus assez intuitif qui peut être déroutant sans les quelques guides dont l'**OOA** prône l'application pour aider l'analyste durant cette démarche:

- La localisation des objets. L'étude s'effectuant généralement dans le cadre d'un projet plus global, il est normal que les premiers objets identifiés se trouvent dans l'espace du problème analysé. Pour cela il est important de puiser dans l'expérience des utilisateurs ou futurs utilisateurs en leur demandant de décrire leurs besoins et leurs attentes vis-à-vis du système, puis de chercher dans l'expérience bibliographique du domaine pour appréhender les terminologies du domaine étudié et nommer correctement les objets.

– La forme des objets à rechercher. Ils peuvent être des structures d'éléments (mécaniques par exemple) dans le domaine traité, classification de fonction, composition de sous-systèmes. Il faut également regarder du côté des interactions externes pour échanger par exemple des informations, et identifier des éléments fournissant ou utilisant des fonctionnalités du système étudié. Les évènements que le système devra gérer peuvent aussi représenter des objets potentiels, il faut les prendre en considération dans l'étude. Les localisations inhérentes au projet comme des adresses mémoire, des lieux géographiques, sont souvent des objets à identifier. L'organigramme des personnes impliquées par le projet peut également fournir des structures intéressantes à stocker comme objet (département, service, groupe de travail).

– Un premier tri est alors à effectuer. Une fois les premiers objets potentiels repérés, il faut évaluer la pertinence de chaque objet pour éventuellement les décomposer à nouveau si les informations gérées par l'objet sont trop hétérogènes, ou regrouper certains objets par exemple si un objet ne semble contenir qu'un seul attribut il peut être pertinent de l'intégrer à un autre objet comme simple attribut.

– L'étape suivante concerne le nommage des objets avec un nom au singulier, même s'il désigne une famille d'objets, éventuellement caractérisés par un adjectif et reprenant le vocabulaire métier du domaine concerné. On peut ensuite représenter l'objet selon le formalisme OOA comme sur la Figure 5.1.

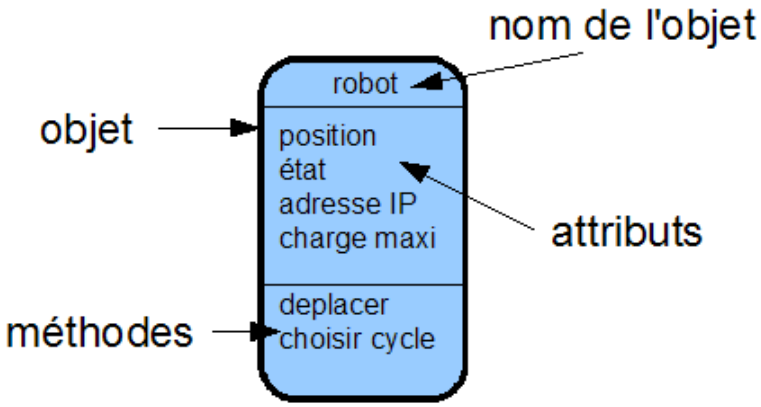


*Figure 5.1.: Représentation d'un objet dans l'**OOA***

5.3.5.1.2 L'identification des structures

Il s'agit ici d'organiser les objets entre eux pour pouvoir élaborer deux grands types de structure, par classification et par composition.

- Structures de classification : on rassemble ici les objets possédant des liens rentrant dans le cadre d'une spécialisation ou d'une généralisation. Si certains objets ont des propriétés ou des fonctionnalités communes, il est intéressant de les regrouper et de proposer un objet qui puisse stocker leurs caractéristiques communes. On établit ainsi des généralisations mais également des spécialisations d'objets.

- Structures de composition : certains objets de taille plus importante peuvent posséder des attributs contenus dans un autre objet sans pour autant en être une spécialisation. On parle alors de composition d'objets, car on *encapsule* des objets entiers à l'intérieur d'autres objets en formant des agrégations. Durant cette phase il est courant de détecter de nouvelles classes lorsqu'une composition se révèle incomplète.

5.3.5.1.3 La définition des sujets

Lorsque le nombre d'objets définis devient important, il peut être difficile d'avoir une vision globale permettant de bien poursuivre les phases de classification. Pour aider les analystes objet cette phase propose d'effectuer des regroupements conceptuels sous le nom de *sujets* pour séparer les domaines abordés par le système et ainsi limiter la quantité d'objets à considérer en parallèle.

5.3.5.1.4 La définition des attributs et des liens

A ce stade on va préciser les attributs contenus dans les objets afin de les décrire de façon plus détaillée. Lorsque les attributs concernent des sous-objets dans une relation de composition par exemple, on spécifie la relation à l'aide des *multiplicités* sur les liens entre les objets concernés. On exprime ainsi, pour une relation donnée, le nombre d'objets qui peuvent être concernés à chaque extrémité de la relation. Ces multiplicités de lien peuvent être de quatre types :

- 1-1, un seul objet de chaque côté de la relation.
- 1-M, un objet peut être lié à 1 ou à une multitude d'objets de l'autre coté de la relation.
- 0-1, il n'est pas obligatoire d'avoir un objet à l'autre bout de la relation, on représente l'aspect facultatif avec un 0.
- 0-M, même cas que précédemment. Lorsqu'une relation existe, elle peut être reliée à plusieurs objets à son autre extrémité.

La Figure 5.2 montre comment on représente graphiquement ces multiplicités dans le formalisme OOA, on parle aussi de *cardinalité*. Cette représentation est proche de celle que l'on retrouve dans la plupart des autres langages liés à des méthodes de conception orientées objet.

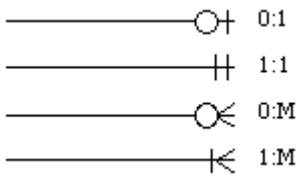


Figure 5.2.: représentation des multiplicités dans l'OOA

5.3.5.1.5 La définition des services

Les attributs ont été spécifiés, il reste à décrire les méthodes attachées aux objets et qui vont représenter le comportement de l'objet, les services qu'il peut offrir au reste du système et aux utilisateurs. Pour faciliter l'identification et la description des méthodes, il est intéressant de se positionner selon trois points de vue par rapport à l'objet en considérant :

- Les actions directes qu'il sera possible d'effectuer sur l'objet (destruction, construction, récupération d'attributs, modification, ...).
- Les calculs réalisés par l'objet

– Les attentes d'évènements, ce qu'on appelle également le *monitoring* .

Ensuite, il est pertinent de considérer l'objet globalement et les différents états dans lesquels il peut se situer pour repérer à quels évènements extérieurs il est susceptible d'avoir à réagir. A ce stade il est donc intéressant d'utiliser des diagrammes de représentations dynamiques (diagramme d'état, de transition, d'évènement).

Une fois les services spécifiés, on peut détailler les échanges de messages entre les objets dans le cadre du système étudié, en représentant les connexions entre objets et/ou avec les utilisateurs du système.

5.3.5.2 La méthode Booch

Également citée sous l'acronyme **OOD** (*Object Oriented Design*) car elle en constitue la méthode la plus répandue avec celle de **Coad & Yourdon** [Coad 1991] ou **OOAD** (*Object Oriented Analysis and Design*), elle porte le nom de son inspirateur [Booch 1991]. Avec ses vues logiques et physiques, elle a été d'abord mise en place pour rationaliser le développement d'applications **ADA** puis **C++** au Département de la Défense Américaine (**DoD** *Department of Defense*). Elle se concentrait sur les phases aval de conception en passant la phase d'analyse, préconisant l'utilisation de **IDEF0** (*ICAM Definition Languages*, partie modélisation de fonctions) basé sur **SADT** (*Structured Analysis and Design Technique*). Elle apporte un élément important qui est la notion de « *Package* », permettant d'organiser et de structurer les modèles. Elle se rapproche plus du domaine de la conception que de l'analyse d'un système, contrairement à l'**OOA**; nous en présentons les grandes lignes car elle a, comme son homologue pour l'analyse, beaucoup influencé les méthodologies actuelles de conception orientée objet.

Bien que les notations graphiques soient différentes de l'**OOA**, on retrouve des similitudes dans les premières étapes d'analyse et dans les concepts utilisés. Pour Booch, le développement doit être conduit suivant un cycle de spécification itératif (on repasse plusieurs fois par les mêmes phases durant le développement du projet en considérant que les exigences du systèmes évoluent et/ou sont précisées au cours du développement du projet) et incrémental (à chaque cycle on passe à un niveau de détail de plus en plus poussé et la quantité d'informations représentées augmente) avec quatre phases principales comme présenté sur la Figure 5.3.

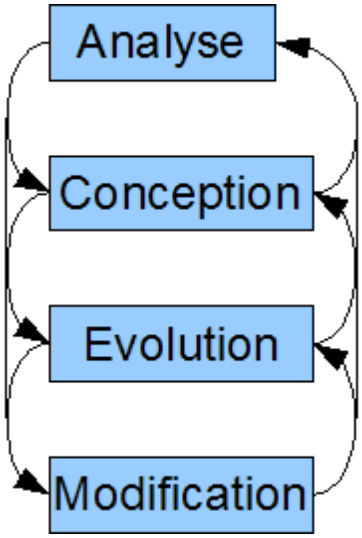


Figure 5.3.: Cycle de spécification pour la conception orientée objet

L'identification des objets et des classes se fait à partir de la description textuelle des expériences des personnes concernées par le problème et de leurs attentes. Comme dans l'OOA, on cherchera à spécifier les actions que l'objet subit et réalise. On intègre un concept supplémentaire qui est *l'interface* de l'objet et qui représente un ensemble de fonctionnalités mises à disposition de l'extérieur. Elle permet de spécifier des contrats de services entre les objets.

Devant la complexité et l'étendue des objets pouvant être mis en oeuvre dans un système, G. Booch a proposé quatre points de vue complémentaires deux à deux et permettant de structurer les spécifications de façon plus claire.

- Un point de vue Logique et Physique
- Un point de vue Statique et Dynamique

5.3.5.2.1 Vues Logiques et Physiques

Elles s'appuient sur quatre types de diagramme et différencient ce qui rentre dans le cadre de l'analyse du problème (modèle logique) de ce qui appartient à son implantation (modèle physique) :

– Le **diagramme de classes** sur lequel on retrouve la définition des classes sous forme de nuages, comme sur la Figure 5.4. Il sert de modèle aux objets, avec la spécification des relations entre ces classes: relations de composition, d'héritage, d'instanciation.

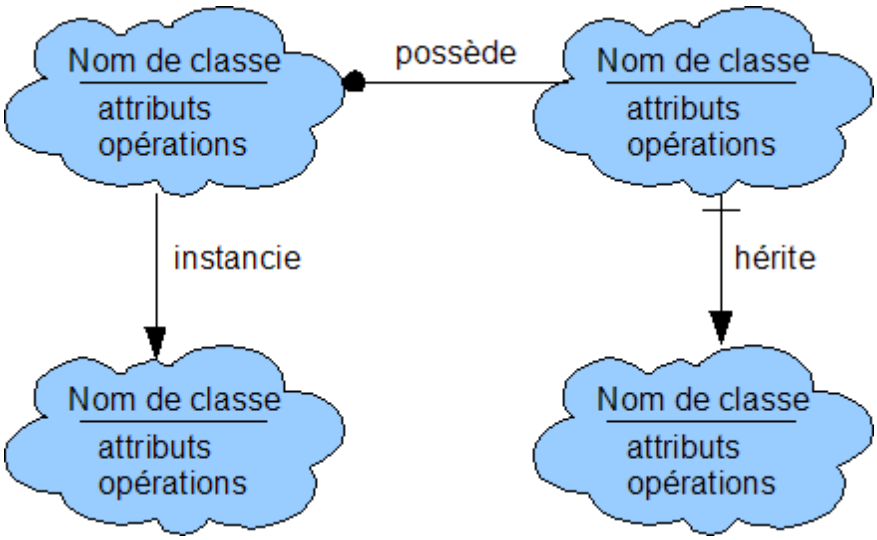


Figure 5.4.: Diagramme de classes dans la méthode Booch

Booch introduit également la notion d'utilitaire de classe qui est une classe liée directement à des mécanismes d'implantation et n'appartenant pas nécessairement au cadre de l'analyse du projet; ce sont par exemple des fonctionnalités proposées par un environnement d'exécution (accès à une base de données, affichage de messages). Un utilitaire de classe est représenté par un nuage avec une ombre portée.

Les catégories de classe sont un autre concept important qui permet de rassembler les classes appartenant à un même contexte de développement donné. Elles permettent de structurer la classification des classes et peuvent être représentées sur un diagramme de plus haut niveau ce qui améliore la lisibilité de l'architecture globale, on parlera de *packages* de classes.

Dans la méthode Booch, d'autres concepts objets plus poussés ont également une représentation particulière. Du fait de la classification en catégories, il est possible de préciser si une classe est utilisable dans d'autres catégories ou non en définissant la *visibilité* des classes (privée, protégée ou implémentée). On a également accès à la

représentation de classes paramétrées appelées patrons de classes (*templates*) pour exprimer le fait que certains types des attributs de la classe ne sont pas connus lors de l'implémentation du système mais sont déterminés pendant son fonctionnement dynamique.

- Le **diagramme d'objets** fait apparaître l'utilisation de classes dans un contexte donné (instanciation). Il permet de représenter les interactions possibles entre différents objets sous forme de liens et de synchronisation de messages allant d'un objet à un autre comme sur la Figure 5.5. Les objets sont représentés comme les classes par des nuages et les listes d'objets sont affichées sous forme d'agrégats d'objets.

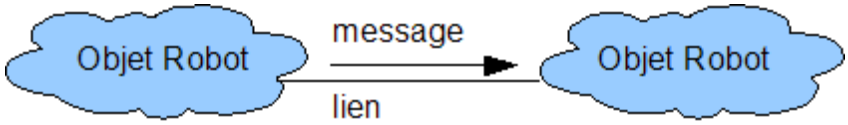


Figure 5.5.: Lien entre deux objets

- Le **diagramme de modules** se situe dans la vue Physique du système; il représente les éléments concrets qui contiendront les informations sur les objets. Dans le cas d'une application informatique, ce seront donc des morceaux de code, (librairies, exécutables) contenant les classes et leurs implémentations. Ce diagramme a été repris en **UML** pour donner le diagramme de composant.

- Le **diagramme de processus** fait également partie de la vue Physique et permet de positionner les modules sur une implantation réelle. Par exemple dans le cas d'une application distribuée, un serveur et un client pourront accueillir des modules précédemment définis. Dans **UML** on retrouve ce concept dans le diagramme de déploiement.

5.3.5.2.2 Vues Statiques et Dynamiques

Elle s'appuient sur deux types de diagramme:

- Le **diagramme de transition** d'états montre pour un objet donné les différents états dans lesquels il peut se trouver et les conditions nécessaires pour

passer d'un état à un autre. On le retrouve comme diagramme d'états en **UML**, que nous présenterons plus loin dans ce rapport.

- Le **diagramme de temps** dont le concept est présent en **UML** sous la forme du diagramme de séquence et que nous aborderons également ultérieurement permet de représenter les différents objets avec des lignes de vies verticales ainsi que l'ordre temporel des messages transmis entre les différents objets pour un processus donné.

5.3.5.3 La méthode HOOD (*Hierarchical Object Oriented Design)*

Mise au point au sein de l'Agence Spatiale Européenne [AES 1989a], [AES 1989b], [AES 1991], la méthode **HOOD** propose une approche intuitive de décomposition hiérarchique. On retrouve la plupart des concepts vus précédemment dans la méthode **Booch**, notamment la notion de module qui est un point important dans la méthode **HOOD**. Le principe est d'identifier non pas des objets directement mais plutôt dans un premier temps des structures plus larges, puis de reprendre ces structures une à une afin de les détailler et de les décomposer à leur tour en structures de niveau inférieur.

La complexité de la première phase est réduite par rapport à d'autres méthodes où l'on identifie dés le départ des objets à partir de spécifications textuelles, par contre les structures identifiées sont très liées au contexte d'étude. Le risque est donc de spécifier des objets (qui soient complètement liés à la structure parente) sans le recul d'une vision globale, ou de définir des objets qui devraient être groupés ou complémentaires à plusieurs endroits dans différentes structures.

Il est facile de démarrer dans cette méthode car on sépare clairement les aspects du système dés le départ, et l'on suit méthodiquement une logique de découpage de la complexité du problème ciblé. On peut ainsi répartir facilement les tâches de spécification en fonction des corps de métier.

Une originalité importante de la méthode **HOOD** au niveau graphique est la possibilité de montrer la correspondance entre un service proposé par un module de plus haut niveau et un service fourni par un module interne, on voit ainsi qu'une interface fournie par un module et comprenant un ensemble de services est en fait implémentée par

des sous-modules et leurs propres services comme sur la Figure 5.6.

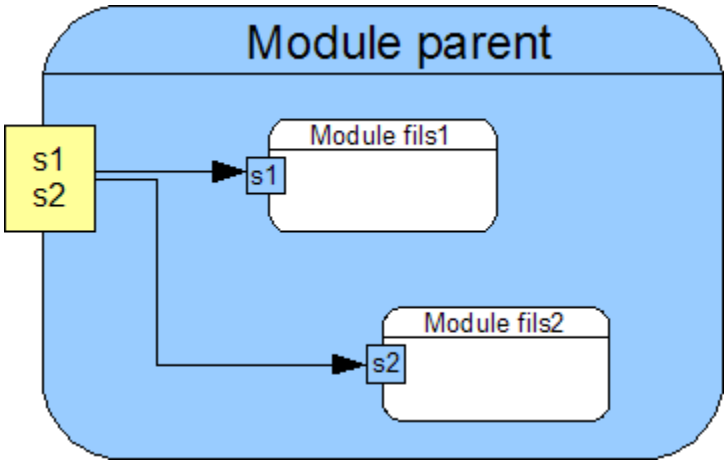


Figure 5.6.: Décomposition des modules et description de l'implémentation des services dans la méthode HOOD

On retrouve une vision similaire dans le langage UML avec les diagrammes de structure composite dans la version 2.1 du langage.

Le déploiement physique des modules est ensuite réalisé avec la représentation de noeuds virtuels représentant des plates-formes d'exécution ou d'accueil pour les modules précédemment définis.

5.3.5.4 L'OMT (*Object Modeling Technique*)

Initiée par James Rumbaugh [Rumbaugh 1991] avec, comme dans la plupart des méthodes objet, la séparation entre les vues statiques, dynamiques et fonctionnelles d'un système, cette méthodologie est née dans les services de recherche et développement de General Electric aux États-Unis. C'est certainement le modèle qui a influencé le plus le langage **UML** car on y retrouve le modèle de classe dans son aspect graphique final sous **UML**, avec la possibilité de représenter les hiérarchies de classes et les relations entre elles (relations binaires et n-aires, multiplicités) d'une manière assez proche du formalise Entité-Relation que l'on avait dans la méthode **MERISE**. Au niveau dynamique on retrouve la description graphique de scénarii mettant en jeu les instances des classes définies dans un diagramme d'états.

Au niveau du langage on retrouve des éléments similaires avec les autres méthodes, à savoir :

- Un Modèle Objet (**MO**) permettant de définir de façon statique les objets avec leurs attributs, leurs méthodes et les relations de généralisation ou de composition entre les objets.

- Un Modèle Dynamique (**MD**) qui définit le cycle de vie des objets en précisant, leur comportement interne avec leurs états successifs, les interactions avec la gestion temporelle des évènements et des messages que les objets peuvent recevoir ou envoyer.

- Un modèle Fonctionnel (**MF**) précisant les fonctions réalisées par les objets au sein de leurs méthodes.

Il faut noter que les éléments graphiques du langage utilisé pour l'**OMT** seront repris de façon quasiment identique dans le langage **UML**, notamment pour le diagramme de classe et d'états.

Si l'on regarde du côté de la méthodologie prônée par l'**OMT**, on est amené à caractériser quatre phases successives pour le cycle de développement. L'approche est pseudo-itérative puisque si l'on se trouve dans l'une des quatre phases, il n'est possible de revenir que dans la phase précédente, contrairement à la méthode **Booch** qui est totalement itérative.

- L'analyse du système avec le développement d'un modèle axé sur les fonctions que le système doit réaliser, sans se préoccuper de la façon dont on veut les implémenter. On réalise un premier MO généraliste avec des classes très générales, puis on implémente un premier MD et MF avant de revenir sur le MO pour y affiner certaines propriétés ou opérations.

- La conception du système, qui consiste en une optimisation, un affinement des MO, MD et MF de la phase d'analyse (et donc décrits de façon très générale) pour obtenir une description plus détaillée qui puisse permettre de lancer une implémentation. On y définit l'architecture globale ainsi que les constituants et les ressources liés au contexte technologique d'implémentation (langage informatique par exemple).

– La conception des objets, qui passe en revue les opérations afin de concevoir précisément les algorithmes qui les composent, en décrivant les structures de données utilisées.

– L'implémentation du système, qui doit mener à la génération dans un langage donné de la logique de contrôle des objets.

5.3.5.5 La méthode OOSE (*Object Oriented System Engineering*)

Aussi connue sous l'appellation *Objectory,* elle a été proposée par Ivar Jacobson [Jacobson 1992] et appliquée dans les développements de projet au sein de la firme Ericsson. Elle couvre tout le cycle de développement avec notamment une méthodologie basée sur les « *cas d'utilisation* » (*Use Cases*), c'est à dire les besoins utilisateurs, avec un affinement de ces cas d'utilisation en scénarii. Elle fut ensuite commercialisée par la société Objectory AB, rachetée ensuite par Rational.

Elle a notamment influencé la norme **UML** au niveau de la spécification des besoins utilisateurs (dans les diagrammes de cas d'utilisation) et de leur prise en compte dans une approche orientée objet.

On retrouve des concepts classiques au niveau structurel pour la représentation et la caractérisation des objets et des modules avec leurs interfaces. L'originalité de cette méthode par rapport à ses homologues se situe au niveau de la soumission itérative de la définition des objets à une validation par les besoins utilisateurs pour s'assurer qu'on ne s'écarte par des objectifs finaux et pour éviter le risque d'obtenir un produit qui ne réponde pas aux attentes client. On a notamment l'intervention d'acteurs externes au niveau du diagramme d'interaction de la méthode **OOSE** comme sur la Figure 5.7, ce qui permet de visualiser directement les échanges entre un utilisateur et les objets du systèmes.

Sur cette illustration, on visualise la spécification des sollicitations (et de leur séquencement) d'un administrateur vis-à-vis d'un système informatique permettant de simuler la constitution d'une palette de produits et la réalisation de ses couches internes.

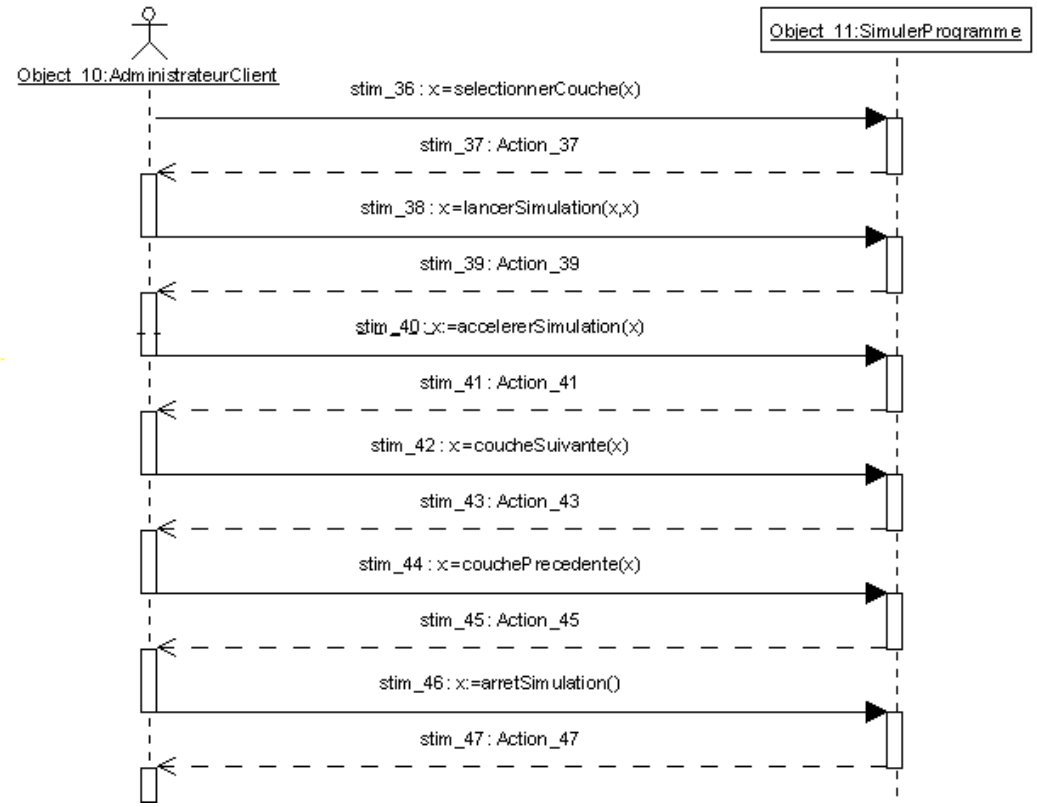


Figure 5.7.: Diagramme d'interaction faisant intervenir un acteur et permettant de prendre en compte les besoins d'utilisation du système

5.4 **Conclusion**

Dans ce chapitre nous avons pu aborder le paradigme objet et évoquer les principales méthodes proposées pour la mise en oeuvre d'un processus de conception ou d'analyse de système en suivant une orientation objet. Ces méthodologies sont fortement liées au domaine de l'informatique car elles sont issues pour certaines directement de travaux dans le contexte d'un langage de programmation donné. Cependant, les concepts abordés peuvent aisément s'appliquer à la définition de systèmes beaucoup plus larges, ne se limitant pas à une application informatique.

Nous avons passé en revue les méthodologies les plus reconnues et insisté sur leurs caractéristiques originales. Aucune n'est universelle, ce sont souvent les caractéristiques du contexte, comme la taille du projet et les choix technologiques réalisés

dans le cadre de son implémentation, qui régissent le choix d'une méthodologie particulière. Nous avons préparé le chapitre suivant, en soulignant pour chaque méthodologie ce qui a influencé l'élaboration du langage standard **UML**, norme faisant partie d'une vision de la conception beaucoup plus abstraite et puissante, l'architecture **MDA** et que nous allons aborder dans la partie suivante.

CHAPITRE 6 :

APPROCHE MDA ET LANGAGE UML

Nous avons passé en revue dans le chapitre 4 les techniques de modélisation que l'on retrouve couramment dans des processus de spécification de systèmes automatisés et nous avons souligné le manque de flexibilité général de ces méthodes classiques d'analyse fonctionnelle ou systémique lorsque le contexte du système évolue. Dans le chapitre 5, nous avons souligné la pertinence de l'approche orientée objet pour la conception d'environnements complexes et changeants, en soulignant également tous les avantages que l'on peut en attendre en terme de capitalisation et de maintenabilité des objets identifiés. De plus, les principales méthodologies orientées objet ont été présentées, introduisant différentes visions du processus de développement de système par un découpage en objets individuels.

Certains acteurs de ces méthodes, devant la multitude de représentations graphiques et de langages associés pour la représentation d'architecture à base d'objets, ont décidé de définir un langage commun à toutes les méthodologies, le langage UML. Ce langage a rapidement fait l'unanimité au sein de la communauté des développeurs en informatique. Son extensibilité avec l'utilisation de profils et de stéréotypes permet de

spécialiser un modèle pour un langage donné et de générer automatiquement un squelette de code exploitable. La modélisation prend donc du recul par rapport à une technologie de programmation spécifique et se rapproche davantage d'une vision objet puriste, permettant une analyse plus propre pour aborder la conception. Nous présentons ce langage dans les paragraphes suivants.

6.1 L' Object Management Group (OMG)

C'est un consortium à but non lucratif initié en 1989 par des industriels désireux de promouvoir l'approche objet dans le monde de la conception, ceci en spécifiant des standards d'échange ou de représentation des données et des systèmes. L'**OMG** est notamment à l'origine de la technologie **CORBA** (*Common Request Broker Architecture*), architecture logicielle permettant la collaboration d'applications hétérogènes virtualisées par l'intermédiaire d'**ORB** (*Object Request Broker*) qui peuvent ensuite communiquer entre eux; on retrouve souvent pour cette architecture l'appellation de modèle **CCM** (*CORBA Component Model*) [OMG CCM 2006].

Initialement concentré sur le développement technique et l'enrichissement de la technologie **CORBA**, l'**OMG** a progressivement recentré ses préoccupations sur l'analyse et la conception d'un point de vue plus global. La constitution de l'**ADTF** (*Analysis and Design Task Force*) rend compte de ce changement de cap avec l'affirmation de nouveaux objectifs:

- Offrir aux concepteurs une meilleure analyse et une plus profonde compréhension d'applications et de systèmes de grande envergure.
- Proposer des architectures possédant des niveaux élevés d'abstraction pour permettre l'interopérabilité des outils de conception et de développement.
- Promouvoir des standards en terme de représentation et de langage de modélisation pour accroître la rigueur et la cohérence des spécifications.
- Mettre en relation des organismes ayant des besoins similaires et les faire participer à la spécification de langages et d'outils standards.

Le processus de mise en place de nouveaux standards suit un protocole rigoureux. Les besoins sont d'abord identifiés lors de rencontres avec des spécialistes dans des domaines particuliers, ces rencontres donnent lieu à des demandes d'informations **RFI** (*Request For Information*) puis à des demandes **RFP** (*Request For Proposition*) de propositions de standard pour répondre à un besoin lorsque la **RFI** est suffisamment pertinente. On peut citer les **RFP** qui ont été menés à terme et ont abouti à la création de standards désormais bien implantés dans le domaine de la modélisation :

- Le RFP ***OA&D Facility*** pour la standardisation des langages de modélisation informatique avec la création d'UML.
- Le RFP ***Meta Object Facility*** pour la standardisation des méta-modèles avec le standard MOF.
- Le RFP ***Stream-based Model Interchange Format*** (**SMIF**) pour la spécification d'un langage d'échange entre outils de modélisation assurant la standardisation de la représentation d'un modèle en XML, donnant lieu à l'adoption de XMI comme standard d'échange de ces modèles.
- Le RFP ***Common Warehouse Metadata Interchange*** (**CWMI**) pour la description des structures de données (bases de données, fichiers de stockage de données) donnant lieu au standard **CWM**.

A partir de ces premiers RFP, de nouveaux besoins ont été soulevés, notamment pour étendre ces premiers RFP à des domaines techniques plus spécifiques :

- Le RFP ***UML Profile for CORBA*** ajoute des stéréotypes de modèles pour **CORBA** et prend en compte le standard de description des interfaces **IDL** (*Interface Definition Language*).
- Le RFP ***UML Profile for Scheduling, Performance and Time*** ajoute des éléments permettant de prendre en compte la notion de temps réel pour laquelle la consommation de ressources et la maîtrise du temps d'exécution sont des points indispensables.
- De nombreux standards sont nés du travail de cette organisation et certains sont même devenus des normes **ISO**. On citera le langage **IDL** (*Interface Definition Language*) ISO/IEC 14750, qui fournit une description pour les interfaces

des composants qui vont communiquer ensemble, et le standard **MOF** (*Meta Object Facility*) [OMG MOF 2006] ISO/IEC 14769 de spécification du stockage des objets pour leur échange entre différentes applications de développement d'objets.

6.2 **Model Driven Architecture (MDA)**

6.2.1 Introduction

Deux contraintes principales empêchent les méthodologies de conception classiques d'obtenir un degré de généricité important, les plates-formes de déploiement et les langages d'implémentation. Une plate-forme est en général le conteneur qui va accueillir les éléments développés dans un langage donné. Cette plate-forme est donc constituée d'un ensemble de services qui permettent aux éléments déployés de vivre et de communiquer entre eux. On appelle cet ensemble de services « *intergiciel* », plus connu sous son appellation anglaise « *middleware* ». Quand on regarde le monde informatique, on se rend compte qu'il existe un grand nombre d'intergiciels disponibles pour réaliser les mêmes services et les phases de conception sont souvent influencées par la nature du *middleware* sélectionné. Un niveau d'abstraction supplémentaire peut être introduit en spécifiant des éléments collaboratifs et la manière dont ils communiquent indépendamment d'une technologie de plate-forme donnée.

Le langage **UML** est la pierre angulaire de l'approche **MDA** de l'**OMG** [OMG MDA 2003]. Il permet la description graphique de tous les aspects d'un système et cela sur différents niveaux de détail. Le fait de stocker des spécifications au sein de plusieurs modèles avec une spécificité croissante permet de garder le maximum d'indépendance vis-à-vis d'une technologie particulière. Ainsi lorsqu'une technologie en remplace une autre, une grande partie du modèle reste valable.

Le développement n'est plus centré sur une technologie finale particulière mais sur un raffinement progressif de représentations abstraites vers des descriptions plus détaillées.

6.2.2 L'Ingénierie Dirigée par les Modèles (IDM)

DÉFINITION 5 : INGÉNIERIE DIRIGÉE PAR LES MODÈLES

L'Ingénierie dirigée par les modèles renvoie à l'utilisation systématique de modèles comme artéfacts de conception primaire dans le cycle de vie. L'IDM est un domaine de l'informatique qui met à disposition des outils, des concepts et des langages pour créer et transformer ces modèles.

6.2.2.1 Présentation générale

Aussi connue sous son acronyme anglophone **MDE** (*Model Driven Engineering*) l'**IDM** constitue une démarche plus globale que **MDA**. Une analogie intéressante pour expliquer le principe de l'**IDM** est de considérer l'enchaînement des étapes de conception de modèles comme un processus de production traditionnel de produits dans lequel on va chercher à spécialiser les produits le plus tard possible. Ainsi des modifications de cahier des charges pour un produit donné n'auront d'impacts que sur les phases aval de production et donc sur le moins de systèmes possible. Dans le domaine de l'industrie logicielle, des travaux ont déjà vu le jour en suivant cette logique: on parle de Lignes de Produits logiciels (**LdP**) [Clements 2001] et on utilise des techniques de gestion de production pour optimiser le cycle de développement de nouveaux logiciels. Aussi voit-on le système comme une composition d'éléments jouant des rôles dans des domaines différents (mécanique, électrique, informatique, automatisme).

On part donc d'un modèle initial faisant office de matière première brute dans lequel le niveau d'abstraction est poussé à son maximum pour obtenir un modèle de base à partir duquel on peut démarrer n'importe quel projet envisageable dans une entreprise donnée. Le formalisme suit lui aussi cette spécialisation successive puisque les concepts utilisables dans un modèle donné vont directement dépendre de son niveau d'abstraction et des choix de spécialisation successifs déjà effectués en amont.

6.2.2.2 Le découpage des spécialisations

En se positionnant au niveau du développement d'un système dont la mise en

oeuvre nécessite des spécifications dans différents domaines de compétence, on devine que lorsqu'un certain niveau d'abstraction va être atteint, il va être nécessaire de spécialiser un modèle en plusieurs sous-modèles en les faisant correspondre à une compétence métier particulière. Le raffinement du modèle est ensuite à la charge des personnes spécialisées dans le domaine considéré qui peuvent alors travailler de façon simultanée par rapport aux autres domaines.

Le maintien de la cohérence globale du modèle entre les différents aspects de spécialisation est un point délicat dans l'**IDM**. Elle doit être assurée à deux niveaux :

- Le modèle de niveau d'abstraction supérieur à partir duquel les modèles enfants spécifiques à leur domaine ont été produits (axe vertical dans le raffinement des modèles).
- Ensuite, pour un domaine de spécialisation donné, il ne faut pas que le modèle conduise à la spécification d'un modèle nécessitant l'intervention d'un autre domaine, ce qui signifierait que la séparation des domaines au niveau supérieur a été mal conduite.

6.2.2.3 L'automatisation de l'implémentation

C'est ce à quoi on espère aboutir lorsqu'on a passé les différents niveaux d'abstraction lors du processus de modélisation: obtenir un modèle suffisamment spécifique dans un contexte donné afin d'avoir une description du fonctionnement la plus proche possible de la manière dont elle est implémentée avec la technologie cible.

Il est alors envisageable de tester la logique de fonctionnement au niveau du modèle; plusieurs travaux ont déjà démontré la faisabilité d'une exécution de modèle avant génération et même de la possibilité de compiler des modèles **UML** pour vérifier leur fonctionnement [Pennaneac'h 2001]. Bien qu'on en soit encore loin, cette possibilité ouvre cependant une perspective de normalisation forte de la programmation, avec l'éventuelle arrivée non seulement de nouveaux compilateurs de modèles mais également de plates-formes d'exécution de ces modèles.

A un autre niveau, l'automatisation du processus de modélisation dans la logique **IDM** et **MDA** passe par l'utilisation de transformations de modèles, consistant en

l'application d'un profil technologique donné afin de transformer un modèle abstrait en un modèle spécifique pour une technologie donnée. C'est ce qui est réalisé à des niveaux d'abstraction très bas en **UML** pour la génération de squelettes de code dans différents langages à partir d'un même modèle et en appliquant des plug-ins de génération contenant les informations du profil correspondant.

La transformation des modèles est un concept clé devant permettre la construction automatique d'une logique de fonctionnement spécifique à une technologie donnée à partir d'un modèle fonctionnel plus abstrait. Ce paradigme est au coeur des réflexions actuelles de l'**OMG** pour **MDA**, et l'enjeu de ces développements pour les processus d'analyse et de conception du futur est primordial.

6.2.3 La transformation des modèles

Comme souligné précédemment, la transformation d'un modèle de niveau plus abstrait en un modèle plus spécifique est un concept important dans l'approche **MDA**. Elle consiste à appliquer un ensemble de correspondances entre un formalisme de représentation générale et un formalisme de représentation spécifique à une technologie ou à un domaine donné. Les éléments du modèle sont mis à jour pour un contexte plus spécifique et le concepteur peut soit appliquer une nouvelle transformation pour obtenir un modèle encore plus spécifique, soit travailler directement sur le modèle obtenu.

Dans le guide d'utilisation **MDA**, une succession de modèles est décrite pour permettre un processus de modélisation conforme au paradigme MDA, nous les présentons ci-après.

6.2.3.1 Le CIM (*Computational Independent Model*)

L'objectif premier du **CIM** est de présenter les spécifications du système que l'on veut modéliser. On se concentre donc sur les besoins auxquels doit répondre le système et non pas sur les moyens qui seront mis en oeuvre pour y parvenir. Le **CIM** centralise les informations relatives aux services que devra implémenter le système et implique la participation des futurs utilisateurs du système pour la spécification des besoins et des cas d'utilisation.

Le **CIM** est également un moyen de communication entre les personnes qui vont concevoir le système et des modèles plus précis et les personnes exprimant des besoins sans la moindre idée de la façon dont il faut les implémenter.

6.2.3.2 Le PIM (*Platform Independent Model*)

Dans cette catégorie de modèles on va décrire les scénarii de fonctionnement permettant de répondre aux exigences du **CIM**. On peut y décrire des objets ainsi que les interactions entre ces objets mais dans un formalisme indépendant d'une technologie donnée. On y représente également les comportements internes des objets avec la description des méthodes qui leur sont associées.

Un concept est souvent utilisé à ce niveau: la *plate-forme virtuelle*. Elle possède les caractéristiques de toutes les plates-formes spécifiques vers lesquelles il sera possible de déployer le **PIM**. On réalise donc le **PIM** en se basant sur les propriétés de cette plate-forme virtuelle, s'appuyant quand c'est possible sur un standard pour le domaine concerné.

6.2.3.3 Le PSM (*Platform Specific Model*)

Le **PSM** représente les modèles spécifiques que l'on veut obtenir en phase de réalisation concrète du système. L'obtention du **PSM** suppose qu'une technologie cible ait été choisie et que la correspondance entre les concepts généraux du **PIM** et ceux utilisés dans le **PSM** aient été établis dans un profil spécifique à la plate-forme cible. On peut passer d'un **PSM** à un autre **PSM** en précisant la technologie cible et on doit, au final, être capable de générer une logique de fonctionnement directement exploitable dans la plate-forme cible.

6.2.3.4 Le PM (*Platform Model*)

Dans le **PM** on va retrouver un ensemble de modèles permettant de caractériser une plate-forme. Il sera donc en général fourni par le distributeur de la plate-forme et permettra de représenter les mécanismes de fonctionnement ainsi que les services de la plate-forme de façon précise. Ces spécifications serviront par la suite durant

les transformations en fournissant un ensemble de concepts techniques pour les correspondances avec les concepts de niveau plus abstrait.

6.2.3.5 Le passage du PIM au PSM

Il s'effectue durant une étape appelée *mappage* (*mapping* dans la terminologie anglaise). Les éléments compris dans le **PIM** sont alors analysés de façon automatique et les concepts précédemment définis à l'aide de types génériques et indépendants d'une plate-forme sont remplacés par des éléments spécifiques à la plate-forme choisie. Une vue de ce processus est représentée sur la Figure 6.1.

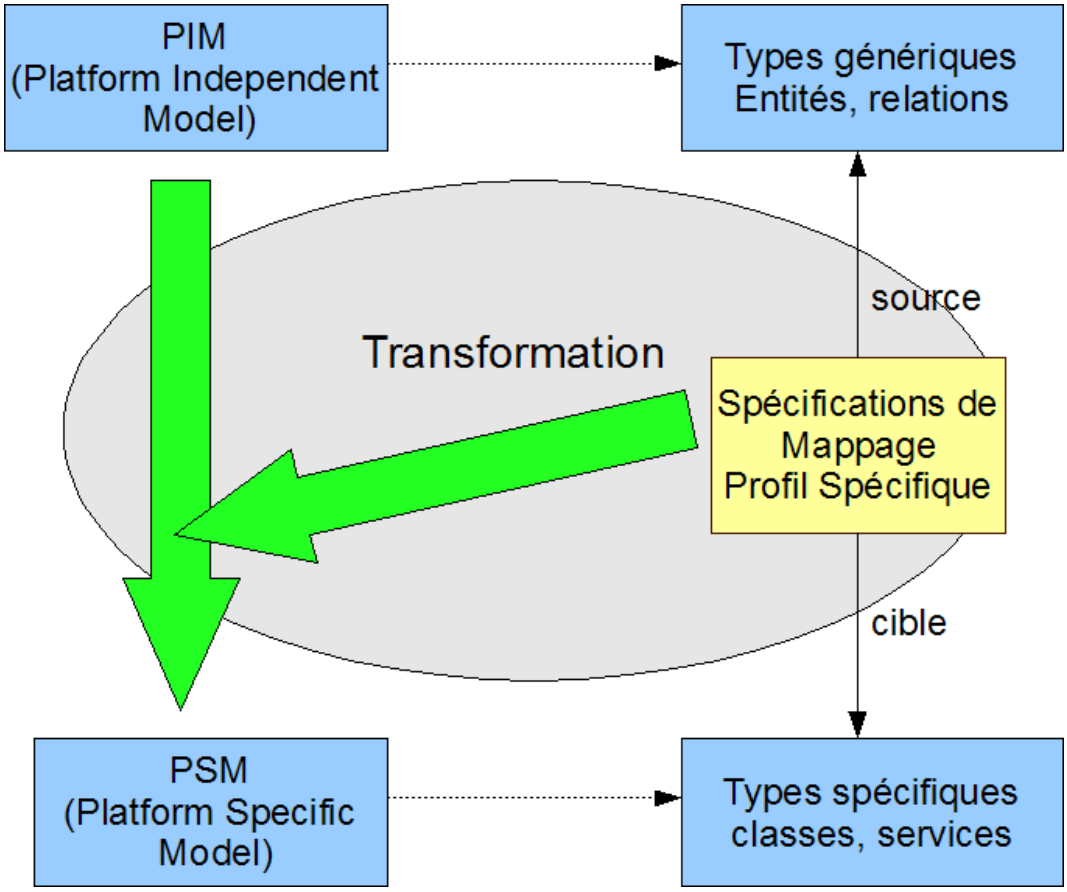


Figure 6.1.: Transformation de modèle selon le paradigme MDA

6.3 Un effort de standardisation à base de méta-modélisation

On pourrait définir la méta-modélisation ainsi:

DÉFINITION 6 : MÉTA-MODÈLE

> *Un méta-modèle est un modèle représentant la structure et les règles d'utilisation d'un langage de modélisation. Il permet de fournir un cadre de construction des modèles dans le langage décrit.*

UML est un langage intégralement défini au niveau de son méta-modèle. Ce même méta-modèle est également construit dans un langage de méta-modélisation spécifié par le **MOF** (*Meta Object Facility*) [OMG MOF 2003]. Il est important de noter pour la cohérence du travail de l'**OMG** qu'il n'existe qu'un seul méta méta-modèle, à partir duquel ont été définis les méta-modèles **UML** et **CWM** (*Common Warehous Metamodel*) par exemple. La Figure 6.2 montre la logique de spécialisation de modèles suivant la logique **MDA**, au sein des spécifications de l'**OMG**, les extrémités de l'arborescence découlant du langage **UML** représentent les profils **UML** permettant de passer du modèle à l'implémentation technologique finale.

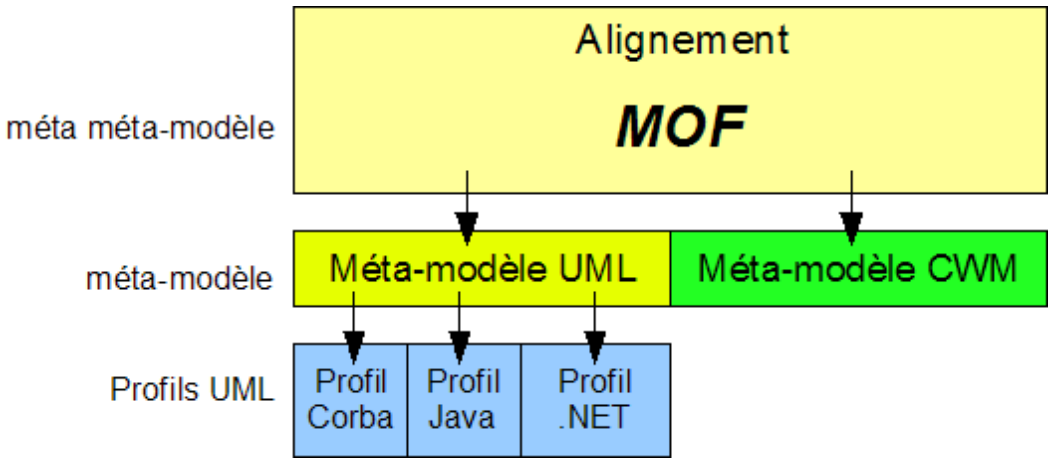


Figure 6.2.: Alignement sur le MOF pour les méta-modèles de l'OMG

On parle également d'architecture à quatre niveaux pour représenter

l'approche de la méta-modélisation prônée par l'**OMG**.

- Niveau méta méta-modèle (**M3**) : le **MOF** est présent à ce niveau comme expliqué précédemment.

- Niveau méta-modèle (**M2**) : on retrouve la spécification des langages de modélisation permettant la représentation des concepts d'un domaine donné. Pour faire l'analogie avec l'approche objet, un méta-modèle est une instance de méta méta-modèle.

- Niveau modèle (**M1**) : on retrouve les modèles tels qu'ils sont réalisés par les utilisateurs des langages de modélisation. Les entités présentes dans ce niveau sont des instances de concepts définis au niveau du méta-modèle correspondant.

- Niveau objets (**M0**) : les éléments des modèles du niveau M1 comme les classes peuvent être instanciés dans un contexte d'exécution donné, sous forme d'objets.

Sur la Figure 6.3 on retrouve cette architecture à quatre niveaux. Au niveau M1, on imagine la modélisation d'une librairie contenant des classes virtualisant un matériel donné, un robot en l'occurrence. Au niveau M0, une instance de la classe Robot est utilisée dans le contexte d'exécution d'une machine.

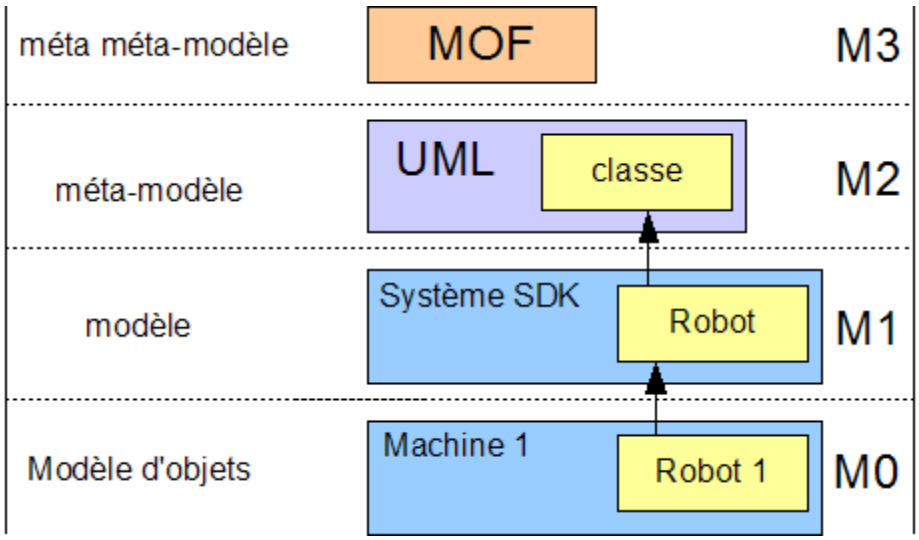


Figure 6.3.: Architecture à quatre niveaux prônée par l'OMG

6.4 Le langage UML (Unified Modeling Language)

6.4.1 Introduction

Dans le secteur du développement logiciel, suite à l'adoption massive de l'approche objet pour la réalisation des applications, et face au besoin pour les développeurs d'avoir à leur disposition de nouvelles méthodologies d'aide à la conception objet, une multitude de méthodes a été lancée dont une cinquantaine entre 1990 et 1995, chacune essayant de s'imposer sur le vaste marché du développement logiciel. En 1995 cependant, répondant à un appel à projet lancé par l'**OMG** (*Object Management Group*) pour l'obtention d'une méthodologie standard de modélisation, trois spécialistes recrutés par la société **Rational Software** et ayant chacun proposé une méthode particulière, décident de spécifier un nouveau langage, comprenant l'impossibilité de s'orienter vers une méthode unique, mais désireux d'obtenir un langage standardisé, ils créent **UML** (*Unified Modeling Language*). Les trois approches qui ont influencé sa spécification et que nous avons présentées dans le chapitre précédent sont l'**OMT**, la méthode Booch et la méthode **OOSE**.

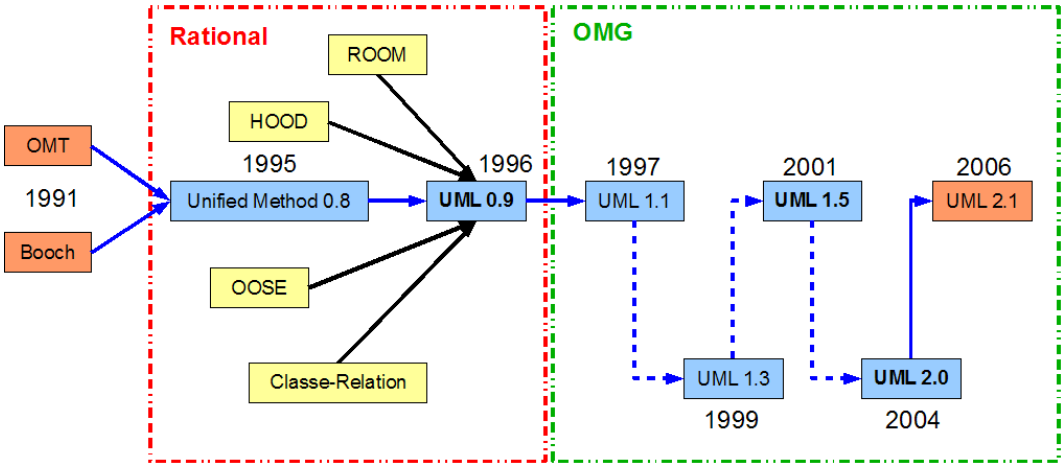


Figure 6.4.: Ligne de vie du langage UML

Jusqu'à sa version 1.5, un certain nombre de lacunes ont été relevées au niveau du langage, Kobryn et Dori [Kobryn 1999], [Kobryn 2001], [Dori 2002], [Ziadi

2004], notamment dans la représentation de structure composite et, au niveau dynamique, dans l'utilisation des opérateurs logiques [Chiron 2005] que l'on retrouve dans tous les langages de programmation. De même la communauté de chercheurs travaillant sur les **MAS** (*Multi-Agent Systems*) a proposé des évolutions possibles du langage, réunies dans l'extension de langage **AUML** (*Agent Unified Modeling Language*) pour pouvoir prendre en compte les spécificités des architectures à base d'agents [Bergenti 2000], [Odell 2000], [Odell 1999], [Parunak 2001].

D'autres domaines ont également proposé de nombreuses extensions au langage pour répondre à leurs propres besoins, comme pour la modélisation des systèmes temps réels: la proposition de **RT-UML** (*Real Time UML*) [Selic 1998], introduisant le concept de capsule et de port qui a donné ensuite lieu à une succession de travaux de thèse et d'articles sur des extensions **UML** pour les systèmes embarqués et les contraintes très particulières de ces domaines. [Gerard 2000], en est un excellent exemple avec la méthode **ACCORD/ UML** puis le projet **AIT-WOODES;** les travaux, conjointement aux autres, ont permis l'amélioration et l'enrichissement du langage **UML** avec la sortie de la récente version 2.0, révisée en 2007 [OMG UML 2007b] et version sur laquelle nous nous appuyons dans la suite de la description du langage.

Cette version est une profonde refonte de la norme; de nombreux concepts qui n'avaient pas de signification ou une cohérence assez vague dans le méta-modèle par rapport aux autres éléments ont désormais été proprement spécifiés. C'est le cas par exemple des collaborations et des composants avec des évolutions que nous explicitons plus loin.

6.4.2 Les diagrammes UML 2.0

Dans les spécifications du langage on distingue deux grandes catégories de description pour un système donné, on retrouve ces deux visions dans certaines méthodes qui ont inspiré les premières versions du langage :

- Une ***description structurelle***, permettant la représentation des éléments d'un système de façon structurée et en offrant une vision statique.

- Une ***description comportementale***, dans laquelle on retrouve les

spécifications dynamiques à la fois internes aux objets identifiés et décrits dans la partie structurelle mais également les interactions entre ces différents objets.

Aucun guide d'utilisation n'est fourni avec l'ensemble des diagrammes dont l'utilisateur peut se servir. C'est l'une des principales difficultés pour le concepteur qui doit choisir de façon pertinente la bonne approche qui convient au cadre de son projet, mais c'est aussi pour le langage un gage d'universalité dans la mesure où il n'est pas influencé par une méthodologie qui risquerait à terme de l'orienter vers des domaines spécifiques comme l'informatique. Pour une approche orientée objet il faut donc se tourner vers des méthodes selon le contexte d'utilisation, voire les mixer pour obtenir une modélisation cohérente.

Nous présentons dans les paragraphes suivants les caractéristiques du langage UML en insistant notamment sur les nouveautés de la version 2.0, sans pour le moment donner de propositions d'utilisation, ce que nous ferons plus tard, dans le chapitre suivant dans le cadre des systèmes automatisés.

6.4.3 Modélisation de la structure du système

Dans cette partie du standard sont décrits les concepts permettant de caractériser la structure d'un système ainsi que les méta-modèles régissant le cadre d'utilisation de ces concepts au sein des diagrammes. Les concepts sont regroupés en *paquetages,* plus communément utilisés sous le terme anglophone *packages* (que nous utiliserons dans la suite), et sont décrits dans des ***diagrammes de structure*** dont les variations permettent de définir des diagrammes plus spécialisés en fonction des éléments que l'on souhaite représenter (classes, composants, structures composites, packages, objets).

6.4.3.1 Les classes

Entité la plus utilisée dans les processus de modélisation, elle permet de stocker les caractéristiques d'un objet donné en définissant ses attributs et ses méthodes, mais également de préciser les interactions avec l'environnement extérieur avec la spécification d'interfaces d'échange, déclarant des services que l'objet fournit ou requiert

lors de son fonctionnement. Le ***diagramme de classes*** reçoit les représentations de ces classes et permet également de lier les classes entre elles avec des relations conformes au paradigme objet: la généralisation, la composition, un exemple est donné sur la Figure 6.5:

Un module physique est représenté par une classe *CPreparateur* avec un formalisme graphique proche de celui utilisé dans la méthode **OMT**, et possède plusieurs relations de composition avec d'autres classes virtualisant des éléments physiques. On visualise ainsi quels matériels peuvent être assemblés au sein du module. Ces entités ont en plus des caractéristiques communes regroupées dans une classe mère *CEltPhys* et avec laquelle ils possèdent une relation de généralisation.

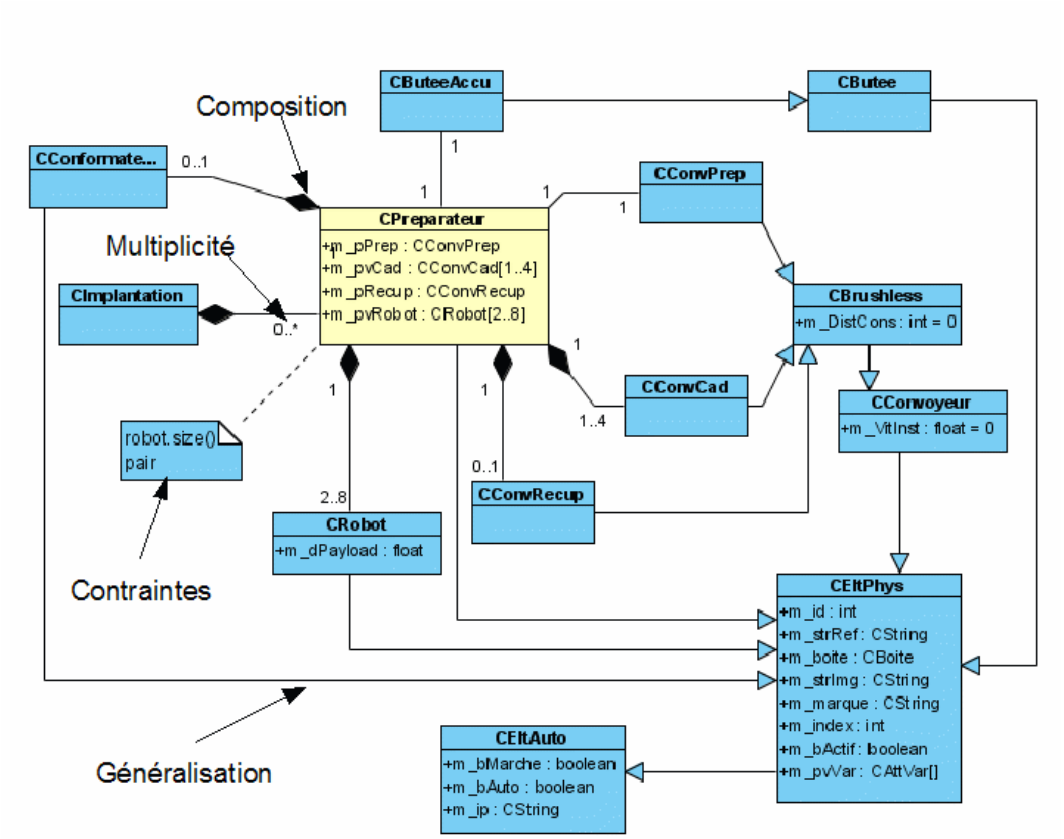


Figure 6.5.: Diagramme de classe de description d'une librairie d'objets matériels

Le diagramme permet également d'exprimer des contraintes diverses en

OCL comme la parité obligatoire du nombre de robots pour notre module physique. Enfin, des multiplicités précisent les relations de composition en fixant des limites de cardinalité. On retrouve le formalisme décrit dans les méthodes qui ont influencé **UML** et nous ne rentrerons pas dans le détail du langage graphique.

Une fois les classes reliées entre elles à un niveau structurel, il est possible de les préciser individuellement par l'intermédiaire d'***interfaces*** d'entrée/sortie. Ces interfaces sont des concepts **UML** qui héritent désormais des classes **UML**. Elles possèdent donc des propriétés et des méthodes. On les utilise pour préciser les ***services requis*** par une classe pour qu'elle puisse fonctionner correctement, ainsi que les ***services fournis*** qu'elle met à disposition du reste du système. La Figure 6.6 contient les spécifications d'une classe colis possédant une interface requise de services d'accès à des données et une interface fournie avec un service de lecture d'information.

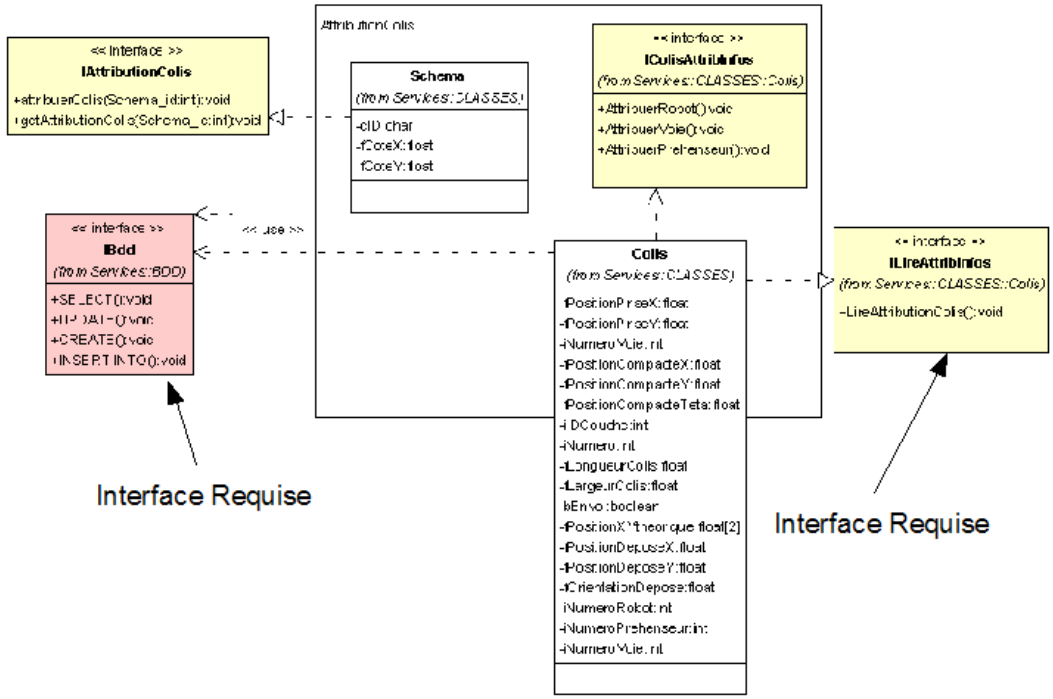


Figure 6.6.: Définition des interfaces requises et fournies d'une classe

Deux autres diagrammes peuvent être utilisés à ce niveau également. Le ***diagramme de packages*** et le ***diagramme d'objets***.

Les packages sont des unités d'organisation du modèle contenant des

diagrammes et des éléments structurels; ils permettent de fractionner le problème en grandes catégories que l'on peut ensuite détailler plus précisément. Les packages ne sont pas indépendants, il est possible pour un élément contenu dans un package d'utiliser ou de dériver d'un élément contenu dans un autre package tant que la relation entre les packages parents est bien définie (inclusion, importation, dérivation ou extension) comme sur l'exemple de la Figure 6.7 avec la séparation des modélisations des phases de fonctionnement d'une machine physique. On précise alors en général le nom du package devant le nom de l'élément, un package représentant également un espace de nommage (*namespace)* un peu à la manière de ceux que l'on définit en informatique pour spécifier l'appartenance à une librairie par exemple.

Le diagramme d'objet permet de spécifier des relations entre instances de classes dans un contexte de fonctionnement donné.

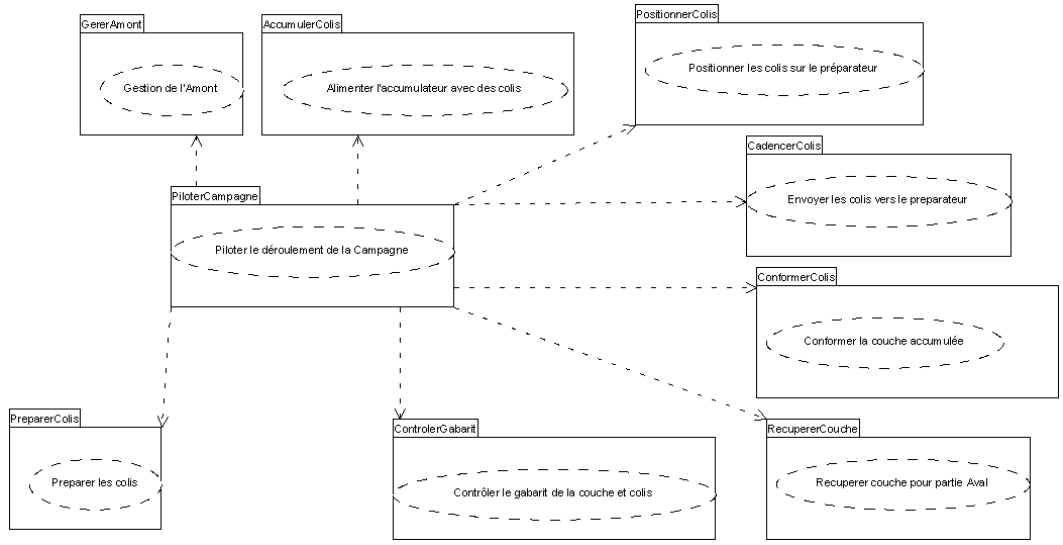


Figure 6.7.: Exemple d'un diagramme de packages.

6.4.3.2 Les composants

Ils héritent directement des classes et représentent des structures de taille variable dont la principale différence par rapport aux classes est qu'ils peuvent être considérés comme des éléments autonomes et ne sont accessibles que via les services de leurs interfaces. On peut les rapprocher au niveau fonctionnel des composants **COM**,

Java Beans, **CORBA** ou **.NET**.

On utilise les ***diagrammes de composants*** qui sont une variation du diagramme de structure pour les spécifier comme sur la Figure 6.8 où l'on a représenté une fonctionnalité de calcul en l'occurrence sous forme de composants en détaillant les interfaces et les services que le composant fournit dans son environnement et ceux qu'il requiert pour fonctionner correctement.

Il faut noter que deux représentations sont possibles pour les interfaces, une représentation par compartiments comme dans nos illustrations et permettant de visualiser en détails les services inclus dans l'interface et une vision simplifiée et symbolique apportant plus de clarté lorsque le diagramme possède un nombre d'éléments important. Nous utiliserons cette représentation simplifiée dans le paragraphe suivant traitant des structures composites.

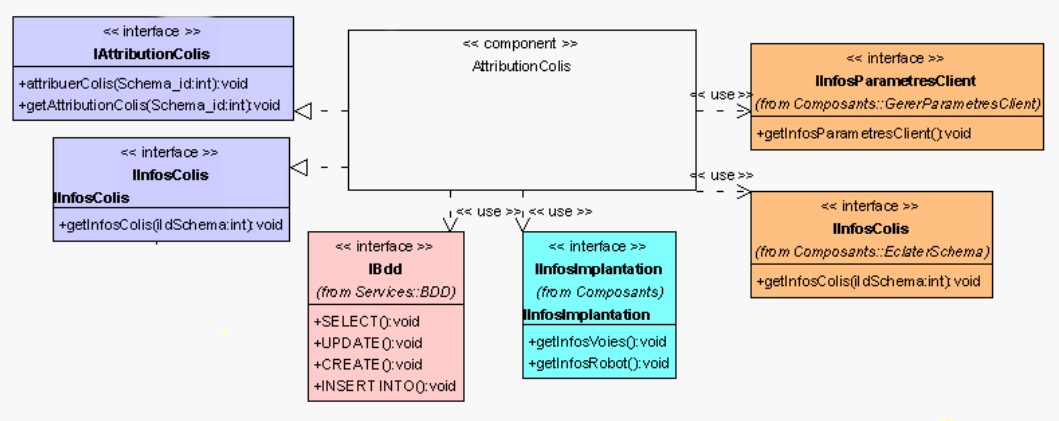


Figure 6.8.: Diagramme de composants

6.4.3.3 Les structures composites

Elles font partie des nouveautés importantes apportées par la version 2.0 du langage **UML**. Elles autorisent une description beaucoup plus détaillée et cohérente de la façon dont les objets et les instances de composants peuvent être encapsulés dans la spécification d'éléments parents mais également des relations entre les services ou méthodes fournis à un niveau d'abstraction supérieur et ceux fournis par les éléments internes.

La modélisation de telles structures se réalise dans les ***diagrammes de structure composite***. Pour exprimer l'isolation de l'implémentation interne d'une structure composite par rapport au reste du système, on utilise le concept de ***port***, qui représente une porte d'entrée/sortie obligatoire pour tous les messages transitant vers l'intérieur ou vers l'extérieur de la structure. Le concept de port a également été enrichi dans la nouvelle version du langage avec notamment le concept de *port comportemental* qui n'est pas lié à l'interface d'un sous-élément mais à une logique dynamique décrite dans un diagramme de comportement (explicité plus loin).

Ainsi l'arrivée d'un message sur ce port entraîne le déclenchement d'un comportement interne spécifié dans une machine à état de protocole.

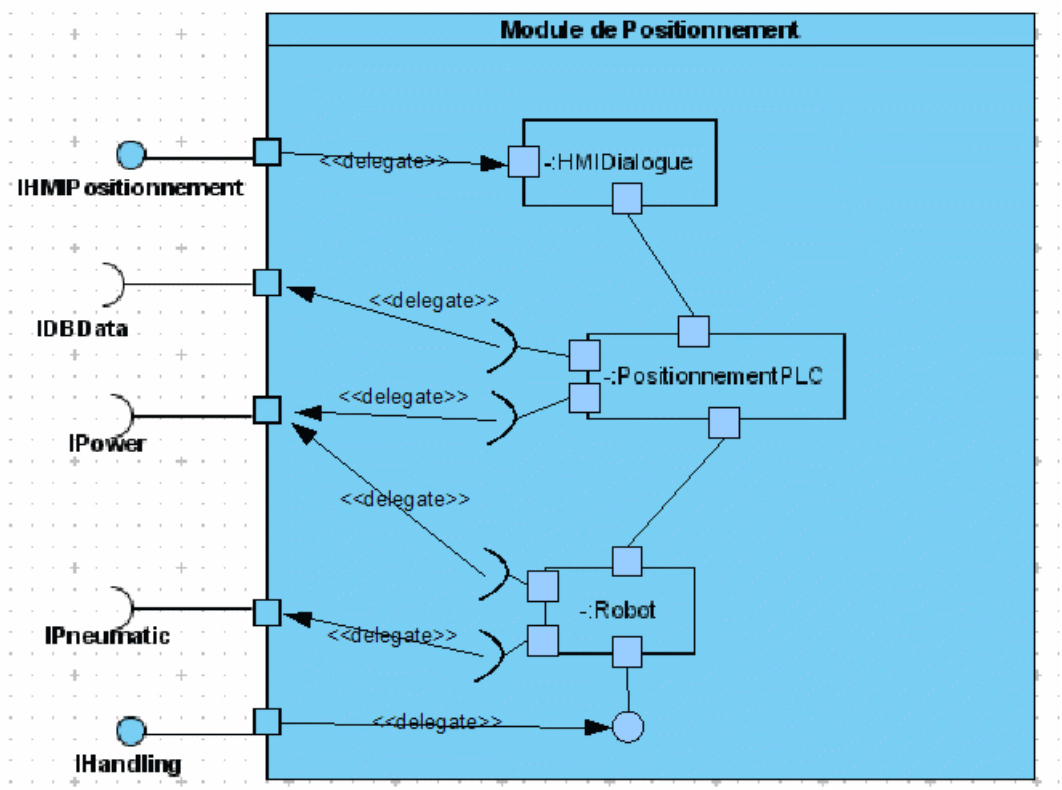


Figure 6.9.: Diagramme de structure composite

Sur la Figure 6.9, un exemple de modélisation en utilisant ce diagramme est décrit. On voit sur la façade gauche de la structure la présentation des ports implémentant des interfaces requises ou fournies. Lorsque ces interfaces sont sollicitées par des éléments extérieurs, les messages sont transmis aux éléments connectés en interne, et inversement dans le cas d'une demande de service provenant de ces mêmes éléments. On appelle *parts* les instances internes représentées sur ce diagrammes; elles correspondent à des instances de classes, de composants définis à l'extérieur ou à l'intérieur de la structure.

La notation simplifiée des interfaces permet une meilleure lecture du diagramme et on obtient, en quelque sorte, un schéma de connexion structurel interne pour un élément donné, en l'occurrence une fonction de positionnement réalisée ici par la collaboration interne d'un robot et d'un automate, avec l'implémentation de services de dialogue avec l'utilisateur dans un module interne **HMI** (*Human-Machine Interface*).

6.4.3.4 Les artefacts et les noeuds de déploiement

Un ***artefact*** est un élément concret d'information produit par le processus de modélisation ou utilisé par le système décrit. Par exemple, les fichiers sources de code informatique qui peuvent être générés dans un langage donné sont des artefacts potentiels, de même qu'un message mail ou un fichier de stockage xml.

Les ***noeuds de déploiement*** UML représentent des ressources de stockage ou d'exécution ayant la possibilité d'accueillir des artefacts ou bien des instances de composant.

On utilise un ***diagramme de déploiement*** pour spécifier les artefacts et les noeuds. Il permet de représenter notamment une architecture physique ou informatique d'accueil pour les composants et les artefacts spécifiés dans la modélisation. Les noeuds peuvent être connectés entre eux formant ainsi des réseaux qu'il est possible de spécifier pour décrire précisément l'architecture du système étudié.

L'exemple de la Figure 6.10 montre un exemple d'utilisation avec différents noeuds sur lesquels on a positionné des instances de composants pour expliciter comment le déploiement est effectué au niveau des différentes plates-formes cibles. Les connections entre les noeuds virtualisent les différents réseaux de communication et d'échange présents dans l'architecture ce qui permet d'avoir une représentation précise de l'organisation globale du système en phase d'exécution.

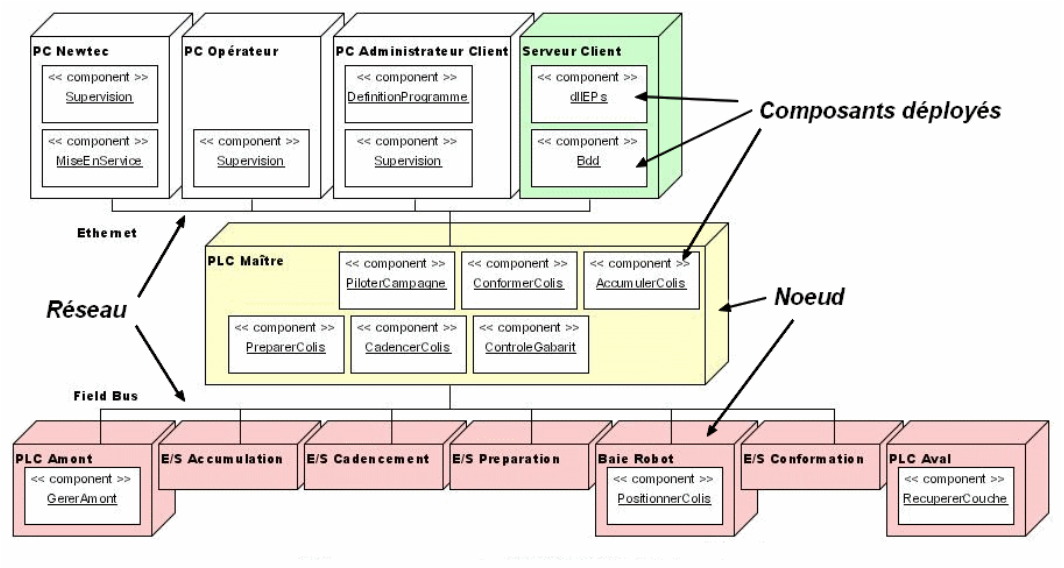


Figure 6.10.: Exemple de Diagramme de déploiement dans le cas d'une architecture d'automatisme classique

6.4.3.5 Les collaborations

Parfois, un concepteur peut vouloir représenter une relation au niveau de la structure sans connaître les objets spécifiques composant la relation. Pour permettre la modélisation de cette relation, UML propose le concept de collaboration. A défaut de pouvoir connecter des objets précis avec la relation, on définit des interfaces pour expliciter des « *rôles* » qui devront être joués par les éléments que l'on voudra connecter par la suite.

L'exemple de la Figure 6.11 permet d'expliquer le mécanisme de représentation. La collaboration, nommée dans l'ellipse en pointillés, est associée à des interfaces à l'aide de relations stéréotypées « *role binding* ». Les interfaces « *IRobot* » et « *IEncoder* » définissent deux rôles qu'il faut absolument remplir pour que le comportement décrit au niveau de la collaboration (qui dérive du concept de classe et peut donc être décrit à l'aide de diagrammes de comportement) puisse se réaliser. L'exemple spécifie une relation dans laquelle un élément (virtualisé par IEncoder) offre la lecture d'une variable de type *float*, et le contrôle d'une vitesse par l'élément Tracking. Sur le diagramme nous avons explicité ces rôles en les connectant à deux objets « *Robot1* » et

« *Encoder* » possédant des interfaces compatibles implémentant les services spécifiés dans les interfaces de la collaboration.

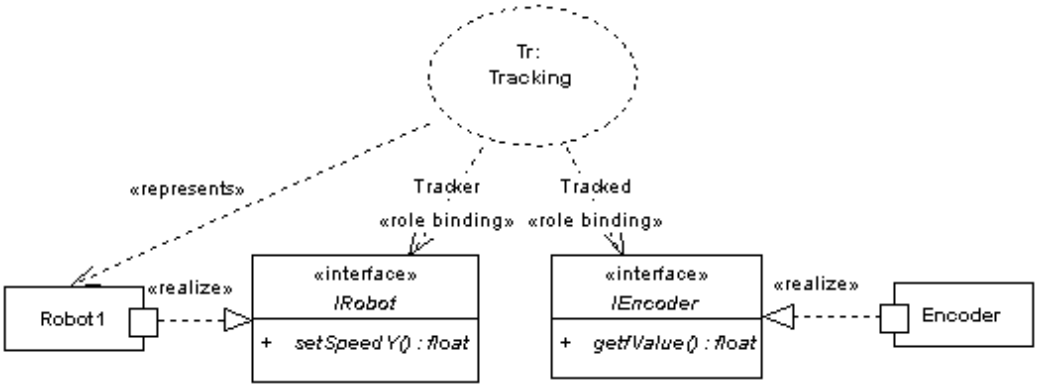


Figure 6.11.: Représentation d'une collaboration dans le cas d'une fonction de « Tracking » Robot

6.4.4 Modélisation du comportement du système

Avant d'introduire des diagrammes, **UML** spécifie en premier lieu des concepts permettant d'exprimer des caractéristiques dynamiques. On fait la distinction entre les actions, les activités, les états, les interactions et les cas d'utilisation. Nous allons passer brièvement en revue leur utilisation pendant le processus de modélisation et indiquer les liaisons effectuées avec le modèle structurel pour maintenir la cohérence globale du modèle.

L'objectif ici est de pouvoir détailler le comportement de tous les éléments décrits de façon statique dans la structure du système et de montrer comment ils interagissent pour réaliser les fonctions attendues de l'ensemble.

6.4.4.1 Les actions

Comme précisé dans le standard, une action représente l'unité élémentaire des spécifications comportementales d'un système. Elle possède un ensemble d'entrées et active ou envoie un certain nombre de sorties en fonction du traitement qu'elle effectue. Tout le travail de description du comportement des éléments structurels va donc consister

à organiser ces actions au sein de diagrammes de comportement de façon à produire le comportement dynamique souhaité pour l'élément étudié.

Un diagramme de comportement est rattaché à un élément structurel (package, classe, composant, objet) et les actions ont accès à tous les attributs de l'élément décrit de façon à pouvoir les mettre à jour ou les utiliser pour déterminer le résultat d'opérations effectuées pendant l'exécution de la dynamique du système.

De plus certaines représentations particulières des actions permettent de modéliser la réception ou l'émission d'informations durant un processus dynamique, ainsi que de spécifier des événements cycliques dans le temps permettant de déclencher une dynamique souhaitée.

6.4.4.2 Les activités

Un peu à l'image des structures composites, une activité contient un ensemble d'actions reliées entre elles qui permettent de réaliser le comportement global de l'activité, sa réaction aux sollicitations extérieures, l'envoi ou le stockage, d'informations. Les activités sont décrites dans des ***diagrammes d'activité***. Lorsqu'une activité est exécutée, les actions internes de l'activité vont être exécutées une ou plusieurs fois selon la façon dont elles sont connectées entre elles et avec les entrées et les sorties de l'activité parente.

On retrouve différentes utilisations des activités selon que l'on se situe dans l'étude d'un flux de contrôle (*control flow*), auquel cas l'activité réagit à des signaux ou à des sollicitations extérieures, ou que l'on se situe dans l'étude d'un flux de données (*data flow* ou *object flow*), auquel cas l'activité possède des objets en entrée et va réagir au changement d'état de cet objet (création, destruction, modification). Sur la Figure 6.12, une fois que l'action « *Valider Opération* » est terminée, l'action « *Envoyer Produit* » est sollicitée, ensuite, c'est l'objet « *produit* » qui est envoyé comme entrée de l'action suivante « *Emballer Produit* » qui, une fois exécutée, déclenche l'action « *Expédier produit* ».

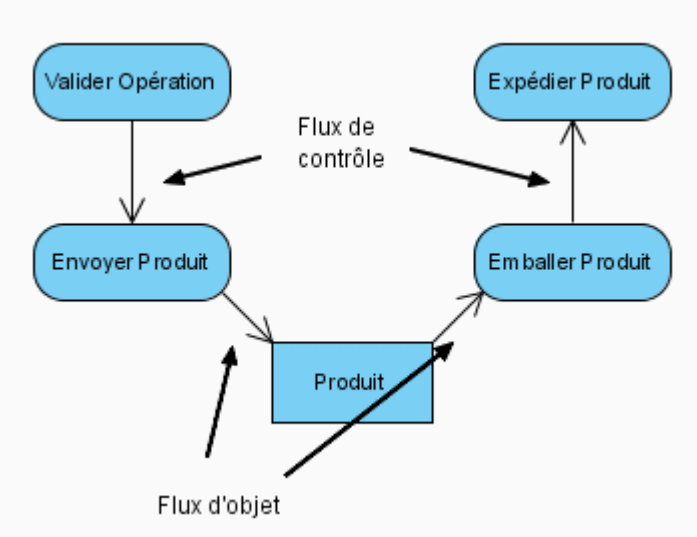


Figure 6.12.: Diagramme d'activité mélangeant flux de contrôle et flux d'objets

Dans un autre contexte de description au sein d'une activité, il est possible de définir des flux concurrents dans le temps pour spécifier le parallélisme de certaines tâches.

A l'inverse des actions qui sont des unités élémentaires de modélisation, les activités peuvent être décomposées en sous-activités, on parle alors d'*activité composite*. Cela permet la réutilisation d'activités déjà décrites à l'intérieur de nouvelles descriptions dynamiques. Par contre des actions comme les activités peuvent être détaillées par des machines à états.

De plus, on distingue les activités basiques des activités complètes dans lesquelles il est possible d'ajouter des contraintes *pré* et *post* locales. Ces contraintes représentent des conditions invariantes pour que le comportement d'une activité puisse se dérouler correctement. Lorsque l'utilisateur introduit des contraintes sur les conditions d'activation d'une activité ou action pour une description dynamique spécifique, il utilise plutôt des noeuds de décision pour caractériser les transitions ou des contraintes directement sur les associations.

Lorsqu'une activité est déclenchée, il se peut qu'elle nécessite des paramètres supplémentaires qu'on ne peut pas intégrer comme objets dans un flux d'objets. On a alors

la possibilité de les représenter sur la frontière de l'activité comme sur la Figure 6.13. L'activité peut alors, en interne, utiliser ces paramètres ou bien les mettre à jour (on peut utiliser une notation graphique fléchée au niveau du graphique lorsque le graphe est complexe et que le nom des paramètres n'apparaît pas sur le rectangle associé). L'activité décrite ci-dessous permet de représenter le comportement d'un robot en mode Tracking, après réception d'un top, il se cale sur une vitesse de déplacement induite par un codeur et effectue un déplacement avant de renvoyer un top et la position courante de l'élément déplacé.

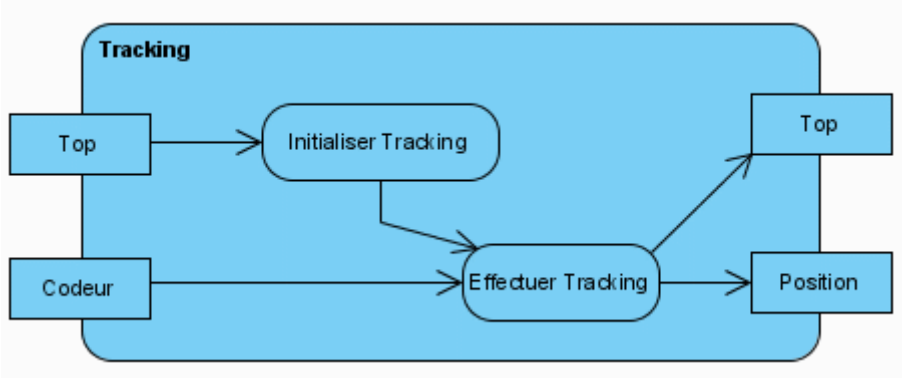


Figure 6.13.: Paramètres d'entrée/sortie pour une activité « Tracking »

6.4.4.3 Les partitions

Pour dire un mot sur les partitions, elles permettent d'intégrer un niveau de description supérieur en regroupant les actions ou activités qui ont des caractéristiques communes par exemple si elles sont réalisées dans un même contexte particulier. Les partitions sont très pratiques, par exemple pour modéliser un flux d'information entre différents services ou sites géographiques d'une société. On partitionne alors selon les services et on intègre les actions de chacun en définissant les flux de contrôle et les flux d'objets correspondant aux échanges entre les différents services (par exemple envoi de mails dans le cas d'un flux d'objets, ou ordre simple dans un flux de contrôle).

Par contre elle n'ont pas de signification dans l'approche objet ce qui peut être déroutant et provoquer des erreurs dans d'autres parties de la modélisation où l'on

risque de ne pas décrire tous les éléments en pensant l'avoir fait en utilisant les partitions, mais il est certain que si l'on reste à un niveau fonctionnel élevé, elles peuvent permettre de clarifier l'organisation générale du système.

6.4.4.4 Les interactions

Alors que les activités et les actions vont souvent être utilisées pour décrire le comportement interne d'un élément structurel, les interactions ont pour vocation de permettre à l'utilisateur de représenter les échanges et les sollicitations qui vont être effectués entre ces éléments structurels. De plus, les diagrammes proposés pour spécifier ces interactions vont permettre d'introduire des contraintes de séquencement entre ces interactions pour que la description du comportement des échanges entre les objets soit précisément modélisée.

Il existe cinq diagrammes décrits permettant la modélisation des interactions d'un système, le ***diagramme de séquence*** (*sequence diagram*), le ***diagramme de vue générale des interactions*** (*interactions overview diagram*), le ***diagramme de communication*** (*communication diagram*), le ***diagramme temporel*** (*timing diagram*) et le ***diagramme de tables d'interaction*** (*interaction tables*). Il n'est pas obligatoire de tous les utiliser dans une même spécification; le choix sera à effectuer en fonction des attentes de l'utilisateur au niveau de la forme de représentation et du type de comportement dynamique qu'il veut décrire.

6.4.4.4.1 **Le diagramme de séquence**

Dans notre application le diagramme de séquence a été souvent utile pour décrire le comportement interne d'une structure composite en terme de collaboration entre éléments internes pour réaliser l'implémentation globale du composant. Son fonctionnement est simple: pour une interaction donnée, on positionne les objets concernés en haut du diagramme en traçant une ***ligne de vie*** verticale sous chaque objet. La description des interactions se fait ensuite en reliant ces lignes de vie avec des éléments graphiques représentant des appels de méthodes, des envois de messages ou de signaux. Lorsqu'une sollicitation atteint la ligne de vie d'un objet, on représente un rectangle fin vertical sur la ligne de vie pour montrer que l'objet est actif et en cours

d'exécution.

On obtient ainsi une séquence dans le temps de sollicitations entre objets. Depuis sa version 2.0, UML propose de plus des éléments graphiques très pratiques, ***les fragments d'interaction***, pour réaliser des opérations simples au niveau du flux de contrôle géré dans la séquence. Ces fragments possèdent un opérateur d'interaction décrivant la façon dont la séquence évolue au sein du fragment. On distingue plusieurs opérateurs d'interaction disponibles dont les plus courants sont :

- L'opérateur alternatif « ***alt*** » dont l'effet est similaire à celui d'une fonction *switch* dans les langages informatiques les plus répandus. Lorsque la séquence atteint le fragment, une condition est testée et le flux de contrôle reprend dans la partie du fragment associée au bon résultat du test de la condition, ou sur une partie par défaut.

- L'opérateur d'option « ***opt*** » permet de définir une condition pour l'exécution d'un fragment du diagramme de séquence. Si la condition n'est pas remplie, le fragment est ignoré.

- L'opérateur d'arrêt « ***break*** » permet d'introduire un fragment de séquence provoquant l'arrêt du fragment en cours et le retour à la séquence normale. Il est donc utilisé à l'intérieur d'un fragment déjà existant. On peut le comparer aux instructions break dans les boucles *for* ou *while* des langages de programmation classiques.

- L'opérateur de parallélisme « ***par*** » permet comme son nom l'indique de diviser la séquence en fragments s'exécutant de façon concurrente et simultanée.

- L'opérateur de boucle « ***loop*** » permet de spécifier un fragment de séquence que l'on va répéter un certain nombre de fois.

- L'opérateur de réutilisation « ***ref*** » permet d'inclure la référence d'une opération ou d'une interaction déjà existante au sein de l'interaction en cours de description.

Les notations du diagramme de séquence ont été fortement retravaillées dans les spécifications 2.0 et on peut désormais aller beaucoup plus loin dans la précision de la description des interactions voire de façon identique à celle dont la logique sera

finalement déployée dans une technologie cible.

Un exemple concret de l'utilisation de ce diagramme dans notre travail se trouve sur la Figure 6.14. L'interaction décrite met à contribution cinq objets: un capteur, un automate, un serveur de base de données et deux robots. Une boucle est représentée pour indiquer la répétition de la lecture de l'état de la cellule. Lorsque la condition de sortie est remplie, un compteur est incrémenté et l'automate vérifie l'identité du robot associée à l'état de ce compteur. Selon la valeur retournée, l'un des deux robots est sollicité avec l'envoi d'une liste de paramètres nécessaires au robot pour qu'il puisse réaliser son cycle. Lorsque ce dernier est terminé, le robot renvoie une confirmation de fin de cycle pour que l'automate puisse mettre à jour les informations correspondantes.

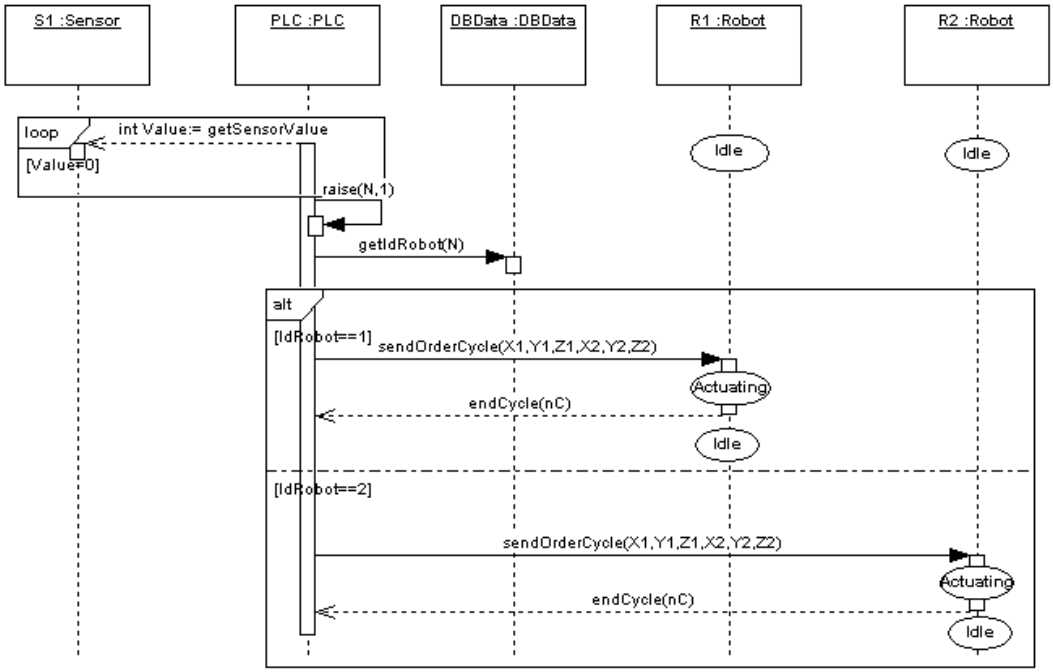


Figure 6.14.: Diagramme de séquence représentant l'envoi d'ordres de missions robot par un automate en fonction de l'état d'un capteur

6.4.4.4.2 Le diagramme de communication

Le diagramme de communication est utilisé lorsque le nombre d'échanges entre les objets est limité et que l'utilisateur souhaite utiliser une vue simplifiée de

l'interaction. Les lignes de vie ne sont pas représentées, on visualise simplement l'envoi des messages entre des objets avec une numérotation des messages permettant de spécifier les échanges prioritaires. On peut représenter la même chose avec un diagramme de séquence.

6.4.4.4.3 Le diagramme de vue générale des interactions

Il permet d'avoir une vue globale des différentes interactions décrites dans les diagrammes de séquence et de l'ordre d'activation de ces interactions qui vont remplacer les actions et les activités dans le diagramme d'activité mais sous une forme non détaillée comme visible sur la Figure 6.15.

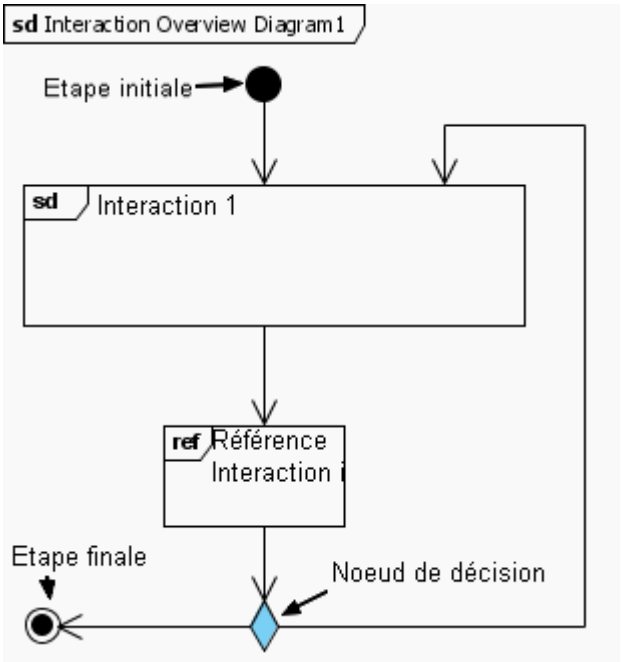


Figure 6.15.: Diagramme de vue générale des interactions

Les interactions référencées ou sous-jacentes sont ensuite décrites dans des diagrammes de séquence classiques.

6.4.4.4.4 Le diagramme temporel

C'est un diagramme un peu particulier permettant de représenter des

interactions liées à des changements d'états. La Figure 6.16 montre un exemple de ce diagramme utilisé pour représenter le comportement d'un objet « *out* »permettant de filtrer l'état d'un autre objet « *in* » en ne prenant en compte que les changements d'état de 0 à 1 considérés comme pertinents c'est à dire quand l'état reste à 1 au moins pendant un temps que mesure un autre objet « *tempo* ». En automatisme on se sert de cette logique pour détecter des fronts montants ou descendants de signaux. Le diagramme temporel permet de montrer les interactions entre ces objets à l'aide de flèches virtualisant l'envoi de messages provoquant des réactions au niveau des autres objets.

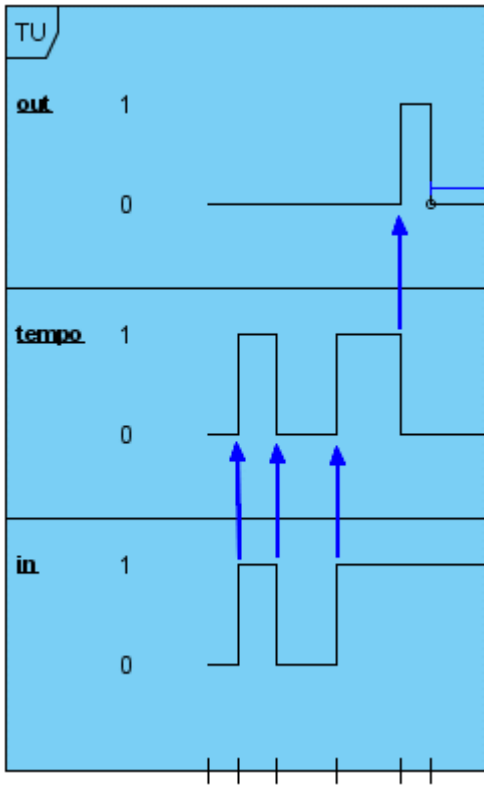


Figure 6.16.: Diagramme temporel d'une interaction réalisant une détection de front montant

On peut noter que ce diagramme est très proche des chronogrammes auxquels les automaticiens sont accoutumés et il permet de visualiser facilement, à un instant t, l'état d'un objet en fonction de l'état d'autres objets participant à l'interaction. Cependant, il devient vite insuffisant dans le cas d'objets complexes possédant un nombre

important d'états intermédiaires.

6.4.4.4.5 Les tables d'interaction

Elles sont à part car ne se basant pas sur une représentation symbolique comme tous les autres diagrammes **UML**. Nous ne le aborderons pas davantage car ce sont des tableaux avec un remplissage textuel qui n'offre pas de cohérence au niveau du modèle global, ils sont simplement décrits dans les annexes de la norme sans autre consigne d'utilisation.

6.4.4.5 Les machines à états

Le diagramme de séquence s'appuie sur une description du comportement basée sur le séquencement des échanges de messages entre objets et le diagramme d'activité centre sa représentation sur un enchaînement d'actions ou d'activités. Les machines à états proposent une approche supplémentaire et complémentaire pour la modélisation en permettant de décrire le comportement *discret* des éléments structurels avec la modélisation des différents états d'un élément ainsi que les conditions et les actions associées au passage d'un état à un autre.

On distingue deux types de machines à états que l'on pourra représenter dans un ***diagramme d'états***:

- Dans leur cadre d'utilisation le plus courant on les définit comme *machines à état comportementales* (*Behaviour State Machines*) reprenant le formalisme de Harel dans ses diagrammes à états (*statecharts* [Harel 1987] ainsi que les possibilités offertes dans la méthode **ROOM** (*Real-time Object Oriented Modeling* [Selic 1994]) pour représenter une architecture de diagrammes d'états sur plusieurs niveaux de décomposition. Ce type de machine à état permet de représenter les différents états du système et la façon dont on peut passer d'un état à un autre en introduisant des conditions sur les transitions entre états. De plus, il est possible au niveau de l'état, de décrire des « *actions d'entrée* » à réaliser avant de rentrer véritablement dans l'état (*entry actions*), des « *actions de sorties* » effectuées avant de passer à la transition sortant de l'état (*exit actions*) et enfin des actions déclenchées pendant que l'état est actif (*do actions*) comme sur la Figure 6.17 qui présente la

description d'un état composant le comportement d'un module de gestion métier pour la palettisation. L'état « *Édition Schéma* » peut également être décrit plus précisément dans une machine à état de niveau inférieur. Il est donc possible de réduire la complexité des diagrammes et de spécifier selon une approche objet le comportement des éléments structurels via ces mécanismes de représentation.

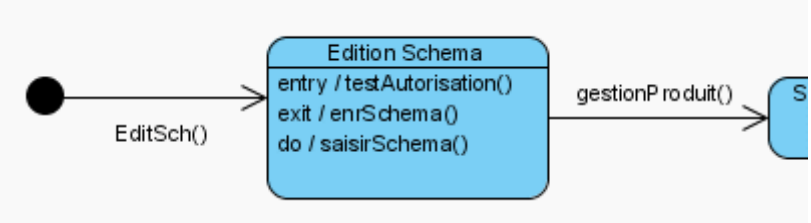


Figure 6.17.: Formalisme d'un état dans une machine à états comportementale

– L'autre type de machine à état est la *machine à états de protocole* dont les états n'ont pas la possibilité de spécifier les trois types d'actions précédents, et qui est utilisée pour spécifier l'enchaînement obligatoire d'opérations et d'états que doit suivre un élément structurel lorsqu'il est sollicité dans une phase de comportement donnée. Ces machines à états de protocole sont le seul moyen de rattacher aux interfaces et aux ports protocolaires des spécifications comportementales au niveau de l'ordre des opérations et des traitements sous-jacents provoqués par une sollicitation extérieure de ces interfaces ou de ces ports.

6.4.4.6 Les cas d'utilisation

Directement influencés par les concepts introduits dans la méthode **OOSE** décrite dans le chapitre précédent, les cas d'utilisation **UML** permettent de définir, dans un ***diagramme de cas d'utilisation***, les interactions sous une forme fonctionnelle entre les éléments extérieurs du système et ce dernier. Ils sont utilisés notamment dans des approches méthodologiques dans lesquelles on part des spécifications client, comme des descriptions textuelles des besoins, et permettent de centrer l'analyse sur ces exigences fonctionnelles.

On peut également les utiliser dans des approches itératives pour vérifier à

chaque étape de la réalisation du modèle que l'on répond toujours aux besoins clients initiaux comme le préconise la méthode **OOSE**. Ils sont un bon moyen d'entamer facilement et intuitivement l'analyse d'un projet en se positionnant du côté des futurs utilisateurs du système.

La représentation est simple, on identifie les *acteurs* externes du système qui peuvent être des personnes ou bien d'autres systèmes complètement indépendants du projet mais ayant à interagir avec le système que l'on souhaite modéliser. Ces acteurs vont être ensuite reliés à des cas d'utilisation virtualisant des situations dans lesquelles le système va échanger avec ces acteurs.

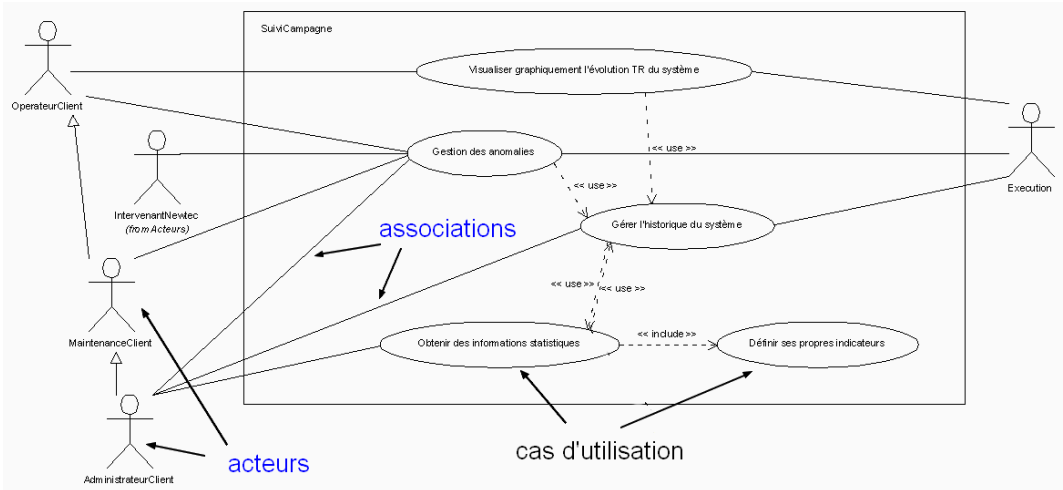


Figure 6.18.: Diagramme de cas d'utilisation

Ces cas d'utilisation peuvent être décomposés en cas d'utilisation avec un degré de granularité inférieur de façon à séparer la complexité initiale des fonctions identifiées. Dans l'exemple de la Figure 6.18, différents acteurs sont représentés sur le côté gauche du diagramme et on peut remarquer la présence de relations de généralisation. Ainsi, un « *administrateur client* » par exemple, aura-t-il accès à un certain nombre de cas d'utilisation mais également à ceux définis pour l'acteur « *père* » dont il hérite, « *maintenance client* ». Ces acteurs sont donc associés à des cas d'utilisation dont on peut également spécifier les liens entre eux à l'aide de relations supplémentaires: utilisation, inclusion, importation ou extension.

6.5 Le langage de contraintes OCL (*Object Constraints Language*)

Le langage **UML** est régi par un méta-modèle définissant sa structure et les règles à respecter pendant la construction des diagrammes. Cependant, lorsqu'il s'agit d'exprimer des opérations simples ou des contraintes, par exemple pour l'expression des conditions sur les diagrammes dynamiques d'**UML** comme les pré et post conditions ainsi que les gardes sur les transitions entre états d'objets, il n'y a pas de directives au niveau d'**UML**. Pour répondre à ce besoin, un langage formel de description de contraintes a été mis en place par l'**OMG**, notamment pour l'enrichissement des diagrammes **UML** en permettant de définir des contraintes et des opérations avec un langage indépendant d'un langage de programmation cible. On retrouve également ce langage au niveau du méta-modèle **UML** pour exprimer les contraintes structurelles de construction des diagrammes et de relation entre les entités du langage.

6.6 La force de l'extensibilité d'UML

Comme nous l'avons vu dans la logique d'architecture **MDA**, pour passer d'un PIM à un PSM, notamment lorsque des choix technologiques sont effectués, il faut alors intégrer des concepts supplémentaires qui seront basés sur des éléments graphiques **UML** spécialisés dans un contexte donné. Cette caractéristique importante d'extensibilité fait que l'utilisation d'**UML** est envisageable dans n'importe quel domaine de compétence, pourvu qu'un profil ait été défini pour la technologie ciblée. Au niveau informatique, ces profils sont nombreux et de plus en plus utilisés (**C++**, **Java**, **Delphi**, **PHP**, **HTML**), mais pour des domaines tels que les systèmes automatisés, il faut s'arrêter à un niveau d'abstraction élevé, faute de profils disponibles à l'heure actuelle.

Cependant, l'implémentation de tels profils serait tout à fait envisageable, car les mécanismes d'extension introduits dans **UML** depuis sa version 1.3 permettent la spécialisation et l'adaptation d'**UML** dans des contextes très éloignés de l'informatique et de nombreux travaux proposent régulièrement des profils spécifiques dans des domaines variés.

6.6.1 Les stéréotypes

Les stéréotypes sont une possibilité donnée à l'utilisateur d'enrichir le méta-modèle UML de façon à intégrer ses propres méta-classes dans un contexte donné. Un lien d'extension est établi avec une méta-classe du méta-modèle comme décrit dans les spécifications de l'infrastructure d'**UML** [OMG UML 2007a]. Il est également possible, au niveau de ces stéréotypes, de définir des propriétés typées (anciennement nommées *tagged values* dans les versions plus anciennes d'**UML**) que les instances de ces méta-classes posséderont automatiquement.

6.6.2 Les profils

Un profil quant à lui va contenir un ensemble de stéréotypes avec leurs propriétés, et constituer une extension par rapport à un méta-modèle de référence comme celui d'**UML** par exemple. Les profils portent en général à la fois le nom du méta-modèle de référence et celui de la technologie cible, par exemple le *Profil Corba pour **UML*** représenté sur la Figure 6.2. La plupart des outils de modélisation **UML** proposent des moyens de définir précisément des profils en introduisant également des contraintes et des règles exprimées en langage **OCL** et conformément aux spécifications de l'**OMG** [OMG 1999].

6.7 XML

XML (*eXtensible Markup Language)* est un langage issu des travaux de standardisation réalisés par **W3C** (*World Wide Web Consortium*) [W3C 2006] initialement pour décrire le code des pages internet. Rapidement ce langage est devenu populaire dans bien d'autres domaines car il permet de représenter n'importe quelle structure de données ou système d'une manière structurée selon un fichier de structure **XML**. Ainsi, pour peu que l'on ait le fichier de structure correspondant, on peut analyser et réinterpréter un fichier **XML** pour retrouver les données initialement stockées. C'est donc un langage de description qui permet d'échanger des informations entre différents environnements, offrant un mécanisme standard de sérialisation de données.

Le langage **UML** s'appuie sur **XML** pour le stockage et l'échange de ses modèles, ce qui permet d'une part pour un utilisateur de passer d'un outil de modélisation à un autre sans que son modèle ne soit modifié, les mécanismes de passage entre **UML** et **XML** étant précisément spécifiés dans le document [OMG XMI 2005], et d'autre part de faciliter la réalisation d'outils de génération automatique de code.

6.8 **Méthode et processus de développement**

Comme précisé précédemment, l'appel à projets de l'**OMG** concernait à l'origine une méthodologie de conception pour les systèmes structurés selon une approche objet, mais **UML** n'est en définitive qu'un langage sans guides d'utilisation pour l'ordre et le choix des diagrammes dans le cycle de développement. Beaucoup de développeurs utilisent **UML** de façon intuitive ou se limitent à certains diagrammes proches de la logique liée à la technologie cible qu'il veulent modéliser. Cependant, on peut citer des méthodologies qui ont essayé de s'imposer dans les processus de conception, notamment le **RUP** (*Rational Unified Process*) inspiré du Processus Unifié décrit par les créateurs d'**UML** [Jacobson 1999] et mis en place et commercialisé par Rational [Kruchten 1999], la méthode **COMET/UML** [Gomaa 2000] [Gomaa 2000b] orientée pour les systèmes concurrents et distribués, **ROPES** (*Rapid Object-oriented Process for Embedded Systems*) [Douglass 1999] [Douglass 1999b] développé au sein de I-LOGIX. Plus récemment on peut citer le projet SPEEDS (SPEculative and Exploratory Design in Systems engineering) [SPEEDS 2006] qui veut définir une nouvelle génération de méthodologies, processus et outils supports pour la conception de systèmes embarqués critiques, le projet DOMINO (DOMaINes et prOcessus méthodologique) [DOMINO 2007] qui propose une démarche basée sur la description d'un système par divers modèles exprimés dans différents langages de modélisation dédiés.

6.8.1 ACCORD-UML

La méthodologie **ACCORD-UML** est particulièrement intéressante. Mise au point dans le cadre du projet **AIT WOODES** [CEA 2003] elle décrit un guide de spécification pour des systèmes embarqués suivant une approche descendante :

- Modélisation d'analyse préliminaire (**PAM**) avec une première analyse des attentes vis-à-vis du système comprenant une identification des cas d'utilisation et une première description comportementale à l'aide de scénarii

- Modélisation d'analyse détaillée (**DAM**) avec une première caractérisation structurelle du système et l'identification de classes et de composants, une caractérisation des interactions entre ces objets puis de leur comportement interne.

- Modélisation de prototype (**PrM**) devant conduire à l'obtention d'un premier prototype en précisant le comportement des instances du système et les contraintes de temps réels qui lui sont liées.

D'autres projets regroupant chercheurs et industriels ont conduit à la proposition de méthodes ou d'extensions pour l'utilisation d'**UML** dans le contexte de systèmes dits temps réels, et répondant à l'appel de l'**OMG** pour le profil ProMARTE (*Profile for Modeling and Analysis of Real-Time and Embedded).*

6.8.2 Le Projet CLIPS

Le projet **CLIPS** dans lequel nous avons été impliqués, il est également question d'une méthodologie de conception des systèmes de production basée sur UML [CEA 2006], [Servat 2005], [Kouiss 2006].

Cette méthodologie est fortement centrée sur le concept de composant et, bien que la première étape de l'analyse soit une analyse classique fonctionnelle avec l'identification des cas d'utilisation, la deuxième phase consiste à identifier des modules physiques réels à partir de l'analyse, sans aller dans le détail de leur implémentation, pour en faire des composants potentiels que l'on décrit de façon classique dans des diagrammes structurels (sous forme de « *boîte noire* » puis « *boîte blanche* ») et comportementaux mais de façon indépendante.

Une taxinomie physique des éléments décrit la couche matérielle dont les éléments matériels peuvent être rencontrés dans un contexte métier donné. Ensuite des composants d'interface sont introduits, les **IEPs** (*Interface Élément Physique*). Ces éléments intermédiaires décrivent notamment deux aspects:

– la description d'interfaces d'interaction pouvant s'intégrer dans les composants de suivi de la partie métier.

– La description du comportement de l'élément physique virtualisé sous une forme proche de celle implémentée au niveau automate.

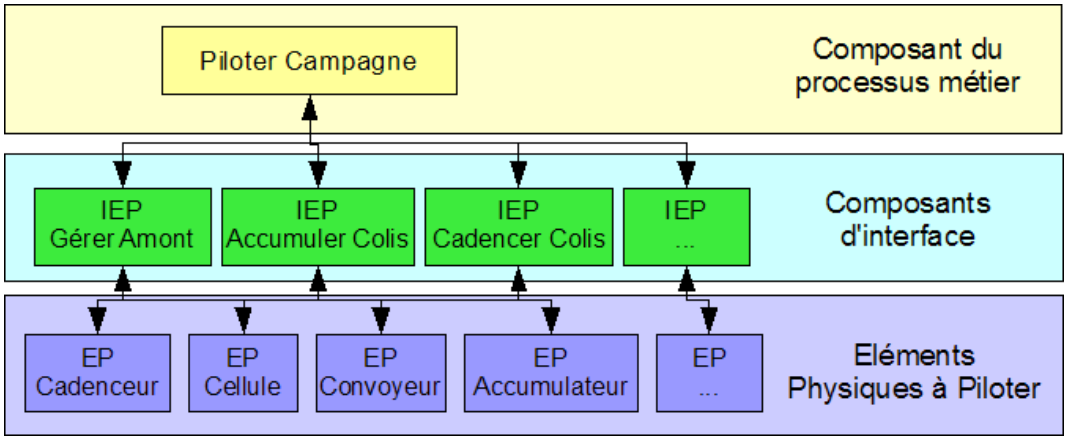


Figure 6.19.: Liaison entre couche métier et couche physique avec les composants d'interface dans CLIPS

La spécification des différents niveaux dont un exemple est donné sur la Figure 6.19 permet de préparer la phase de déploiement vers différentes cibles.

– D'une part la plate-forme logicielle accueillant les composants métiers et apportant des services de haut niveau pour qu'ils puissent échanger entre eux ou avec l'extérieur. La génération a été effectuée pour une plate-forme particulière en s'appuyant sur un « *plugin* » réalisé pour l'environnement **Eclipse** et basé sur **UML2.0** pour la représentation des diagrammes de composants.

– D'autre part la plate-forme accueillant la logique de commande de la couche physique, en l'occurrence un automate. Cette phase de déploiement possible est abordée dans **CLIPS** à notre initiative, mais se limite lors du projet à la description des variables d'échange pour quelques exemples, générées à l'aide d'un autre *plugin* sous **Eclipse** pour donner un listing en **XML**.

Nous avons contribué à l'élaboration de la méthodologie et à la réalisation des composants logiciels sur un prototype. Nous voulions, dans nos investigations, aller encore plus loin dans la spécification du système automatisé cible. C'est pourquoi nous ne

nous attardons pas sur les résultats de ce projet et insistons dans la suite de ce mémoire sur une méthodologie différente permettant d'aller jusqu'à la génération de code d'exécution automate.

6.8.3 Les approches récurrentes

Il existe malgré tout certaines caractéristiques que l'on retrouve dans les méthodologies et qu'il est conseillé d'intégrer dans une approche de conception en **UML** selon l'avis de ses propres créateurs. Un processus de modélisation pour un système peut être :

- Itératif : lors de la spécification du projet il est intéressant d'analyser à nouveau les modèles déjà réalisés pour un domaine donné une fois les autres renseignés, on considère ainsi que le contexte du projet peut évoluer en fonction de ce que l'on définit à l'intérieur. On retrouve la logique du modèle en spirale de Boehm [Boehm 1988].

- Incrémental : en cherchant à chaque itération à aller un peu plus loin dans le détail de la description.

- Centré sur l'architecture: dans ce cas on définit, dés le départ, les différents domaines concernés par le système et la façon dont ils sont reliés entre eux. La méthode **4+1** de Kruchten [Kruchten 1999] est centrée sur l'architecture et spécifie quatre vues à partir desquelles il faut analyser le système: une vue logique (classes, objets et interactions), une vue de réalisation (regroupement dans des composants), une vue des processus (flux de contrôle du système) et une vue de déploiement (implantation cible, ressources matérielles). En outre, Elle fait intervenir une vue supplémentaire au niveau des besoins utilisateurs devant être considérée pour toutes les vues précédentes.

- Piloté par les cas d'utilisation, comme dans la méthode **OOSE,** en partant des besoins fonctionnels et en adoptant une démarche descendante de décomposition structurelle avec, à chaque niveau, une analyse comportementale.

6.9 Conclusion

Ce chapitre nous a permis d'introduire les possibilités de modélisation du langage **UML** ainsi que les méthodologies paraissant les plus intéressantes dans le contexte de modélisation des systèmes de production automatisés avec des diagrammes permettant de représenter des contraintes liées à ce type de matériel.

Devant le nombre de diagrammes disponibles et le fait qu'il est possible de représenter une même description à l'aide de symboliques différentes, l'architecte **UML** peut être dérouté pendant le processus de spécification s'il ne suit pas l'une des méthodologies décrites précédemment. En effet, **UML** est souvent utilisé de façon personnelle et individuelle par les développeurs qui vont se servir de tel ou tel diagramme selon ce qu'ils veulent décrire, par exemple une base de données avec un diagramme de classe, un algorithme de calcul avec un diagramme d'activité, considérant que les méthodes en se voulant génériques et généralistes deviennent trop complexes et trop lourdes à mettre en oeuvre pour de petits projets.

Dans le chapitre suivant nous développons une approche de conception pour les systèmes automatisés basée sur **UML** en introduisant un stéréotype de composant multi-facettes et en détaillant notamment la partie décrivant la représentation de la logique d'exécution.

CHAPITRE 7 :

NOTRE APPROCHE MÉTHODOLOGIQUE

7.1 Introduction

Dans cette partie nous détaillons notre approche méthodologique de conception pour les systèmes automatisés.

Nous avons tout d'abord vu dans une première partie les caractéristiques générales des systèmes automatisés et les différents aspects à prendre en considération lors de leur spécification, ainsi que les technologies actuellement utilisées pour les concevoir notamment au niveau de la logique programmée dans l'automate en utilisant les langages **IEC 61131**.

Dans la partie suivante nous avons introduit l'utilisation de la modélisation comme base du processus d'analyse des systèmes en décrivant dans un premier temps les méthodes de conception traditionnelles principalement utilisées ces dernières années pour la spécification de systèmes de production automatisés. Cette partie nous a également permis de souligner les difficultés rencontrées par la suite pour faire évoluer ces

spécifications lorsque le contexte du système se métamorphose et que les changements opérés dans le système deviennent rapidement complexes, déstabilisant une structure qui n'a pas été pensée pour s'enrichir au cours du temps.

La modélisation est donc tout à fait essentielle dans un processus de conception mais le problème de la méthodologie à utiliser reste complexe et difficile à résoudre pour répondre à toutes les composantes devant être prises en considération dans les projets de développement actuels, dans lesquels les technologies évoluent sans cesse et où il devient dangereux de spécialiser une spécification pour un contexte donné. L'approche objet est une réponse pertinente à cette problématique comme nous l'avons décrit précédemment. Le problème de la mise en oeuvre est cependant encore loin d'être standardisé mais les supports de cette standardisation que pourraient être le langage **UML** et l'architecture **MDA** sont eux bien décrits au niveau de la forme même si leur utilisation reste à l'initiative du concepteur.

Notre travail propose une vision bidirectionnelle du processus d'analyse et de développement centrée sur la modélisation de composants multi-facettes avec le langage **UML** et l'extension **SysML**, selon le cadre architectural **MDA**. Nous avons intitulé cette méthode **MBCSA** (Méthodologie Bidirectionnelle de Conception des Systèmes Automatisés). Les applications pour une facette donnée sont définies dans des conteneurs qui pourraient être assimilés à des *middlewares* mais dans des contextes différents de l'informatique. A partir de ces conteneurs, le déploiement serait possible selon la technologie cible choisie en utilisant des mécanismes de génération automatique comprenant la génération de fichiers de représentation basés sur **XML** dans un format standardisé pour le domaine ciblé.

7.2 L'extension SysML

7.2.1 Introduction

Dans notre méthodologie, nous abordons la description d'une facette automatisme pour les composants que nous décrivons. Les diagrammes **UML** qui y sont

détaillés ont pour objectif l'aide à l'implémentation au niveau de l'automatisme et en l'occurrence, pour garder une architecture à base d'objets, en blocs fonction. On vise donc à obtenir la génération de blocs fonction à partir d'éléments **UML**.

Cependant, en essayant de représenter la forme d'un bloc en **UML 2.0**, on se heurte à une difficulté majeure qui est la représentation des données statiques en entrée et en sortie de l'objet [Heverhagen 2001]. Ces données ne correspondent pas à l'arrivée de signaux ou de sollicitations externes et elles n'appartiennent pas à des éléments internes. Différentes propositions d'extension ont été réalisées :

- Les stéréotypes **FBA** (*Function Block Adapters*) de T. Heverhagen [Heverhagen 03] avec l'introduction d'une représentation personnalisée qui se présente sous une forme hybride entre les blocs fonctions et les classes **UML 2.0**. Bien que difficilement exploitable directement, le travail de Heverhagen soulignait de façon pertinente les insuffisances graphiques du langage dans le cadre de la représentation d'éléments possédant des connexions statiques de variables et il est fortement probable que ses travaux aient influencé le concept de bloc en **SysML**.

- Le projet **CORFU** initié par K. Thramboulidis [Thramboulidis 2004] propose une astuce de représentation pour modéliser ces variables, en les stockant dans des classes particulières (InputData, OutputData) possédant uniquement des attributs pour représenter les entrées et les sorties d'un bloc fonction. La génération à partir du modèle permet d'obtenir un fichier xml puis en **C++** avec l'outil développé lors du projet : **CORFU-FBDK**. Le projet va également un peu plus loin puisqu'il traite des blocs **IEC 61499**, en ajoutant deux classes supplémentaires (InputEvent et OutputEvent) pour lister les évènements d'entrée/sortie d'un bloc exécutable **IEC 61499**. Bien que permettant de générer un code **C++** avec le squelette des blocs, il n'y a pas véritablement de correspondance avec les langages de programmation **IEC 61131**, et l'utilisation de classes pour lister les variables d'entrée/sortie ne permet pas d'obtenir un schéma global pour relier les blocs entre eux.

- L'extension **UML-PA** (*UML for Process Automation*) [Katzke 2005], [Bitsch 2005] est également un travail d'investigation sur les possibilités de générer des applications d'automatisme à partir de spécifications **UML**. B. D. Witsch et B. Vogel-Heuser, à la suite de ces propositions, ont également lancé une collaboration

avec *Beckhoff* et montré qu'avec leur extension d'**UML** ils pouvaient générer un code réutilisable sous l'environnement *TwinCat*. Ce travail expérimental est intéressant et montre qu'il est possible d'effectuer une transition automatique entre une modélisation **UML** et un langage **IEC 61131**. Il souligne toutefois les limites de la norme **IEC 61131** pour l'approche objet avec l'absence de l'utilisation de l'héritage dans la définition des blocs fonction.

Plutôt que d'utiliser une des extensions précédentes ou même d'en proposer une nouvelle, nous exploitons l'extension officielle récente de l'**OMG** pour la modélisation des systèmes, **SysML** [OMG 2006]. Nous décrivons les nouveaux concepts que cette spécification apporte dans [Chiron 2007].

7.2.2 Le stéréotype « *Block* »

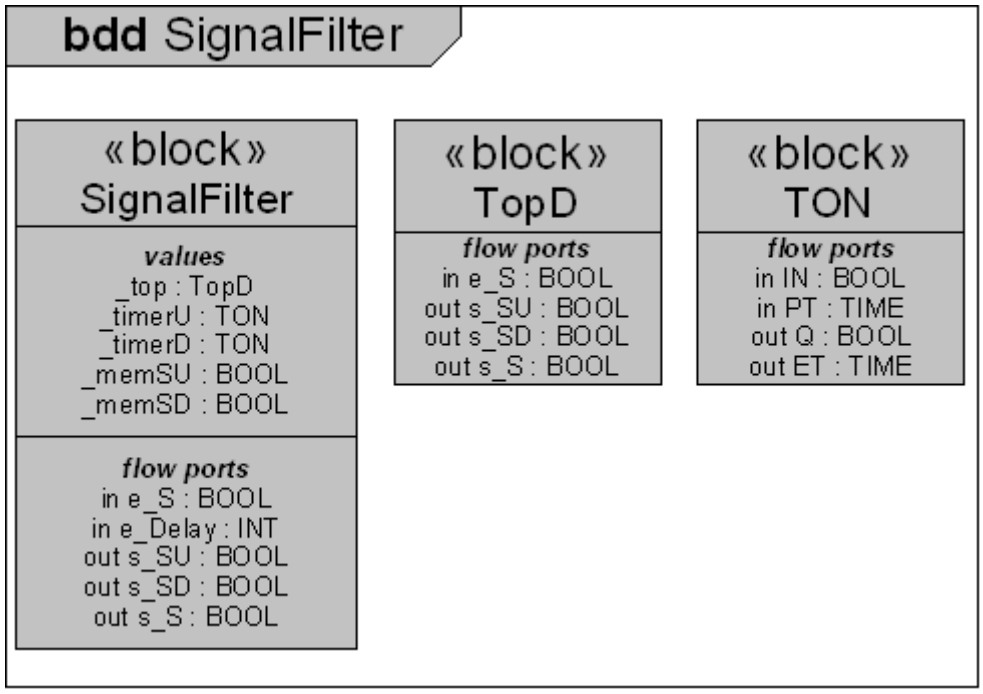


Figure 7.1.: Diagramme de définition de blocs en SysML, vue en compartiments

Pour le représenter, **SysML** introduit un nouveau type de diagramme, le ***diagramme de définition de bloc*** (« *bdd* » *block definition diagram*) comme on peut le

voir sur la Figure 7.1. De plus, il permet également de décrire la structure interne d'un bloc avec le ***diagramme interne de bloc*** (« *ibd » internal block diagram*) de la même façon que dans les diagrammes de structure classique **UML 2.0**. Le stéréotype bloc hérite directement d'une classe **UML 2.0** mais a la possibilité d'accueillir un nouveau type de port, les *ports de flux,* traduction de « *flow ports* ».

7.2.3 Les ports de flux de données

Ces éléments héritent des ports **UML 2.0** et représentent l'extension majeure de **SysML**, avec l'introduction de nouveaux symboles graphiques pour les représenter. Ils peuvent être de deux types, « *atomiques* » lorsqu'ils représentent une seule donnée, ou bien « *non atomiques* » et décrits alors dans une spécification de flux extérieure. Nous nous intéresserons au premier type pour l'analogie avec les variables automates unitaires en entrée des blocs, mais pour des structures plus complexes en entrée, il faudrait soit utiliser un type plus complexe pour le port, soit utiliser une spécification de port contenant les éléments de la structure.

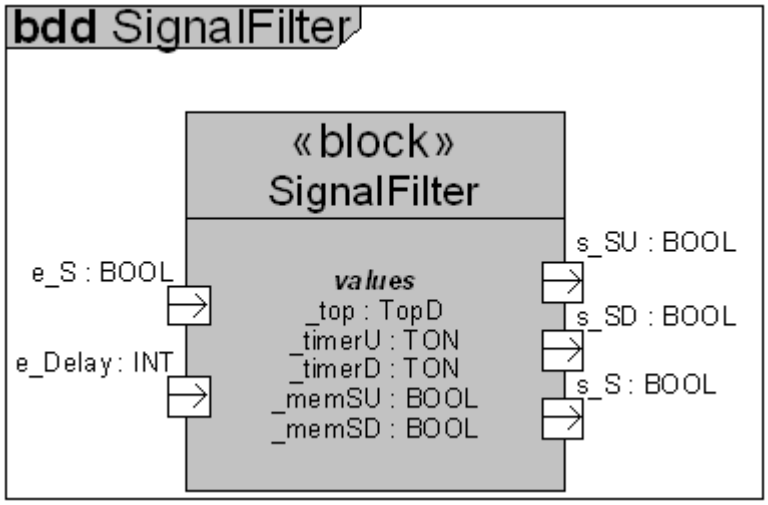


Figure 7.2.: Bloc SysML avec la notation graphique des ports de flux de données

Les ports de flux définissent les données d'entrée et de sortie qui sont reliées directement à d'autres éléments. Ces données sont en quelque sorte partagées puisque lorsqu'un bloc modifie une donnée de sortie, toutes les entrées d'autres blocs reliées à

cette donnée sont automatiquement mises à jour. De plus, ces ports de flux définissent un sens qui peut être de l'extérieur vers l'intérieur (« *in* »), l'inverse (« *out* ») ou les deux à la fois (« *inout* »). L'analogie avec les blocs fonction automates est donc désormais possible sans artifice particulier si ce n'est qu'il faut mettre le méta-attribut (attribut de la méta-classe) « *isBehaviour* » à 0 pour ne pas représenter un port comportemental. L'exemple de la Figure 7.1 montre une représentation par compartiments de ces blocs avec dans la partie « *flow ports* » la description des entrées/sorties typées et avec un préfixe de définition indiquant le sens du flux de données. Le cas décrit, illustrant trois blocs, un filtre de signal « *SignalFilter* », un détecteur de fronts de signal « *TopD* » et une temporisation « *TON* », est expliqué plus précisément dans [Chiron 2007]. La représentation avec la nouvelle symbolique graphique est visible sur la Figure 7.2.

7.2.4 Une vue composite

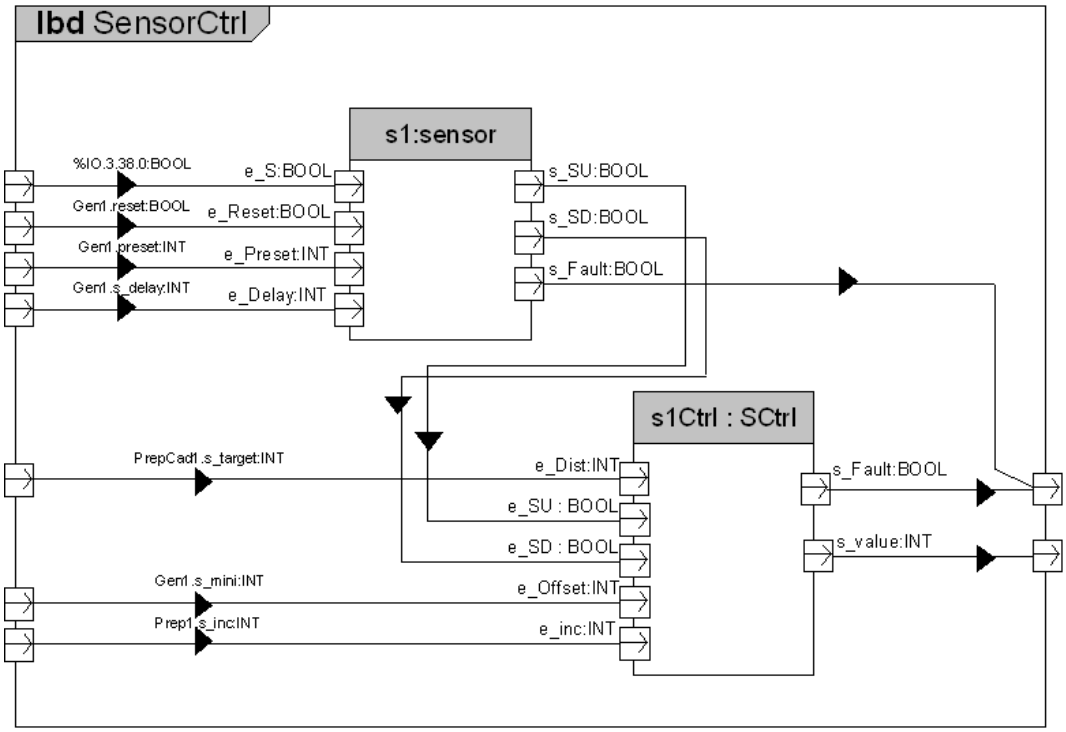


Figure 7.3.: Diagramme interne du bloc « SensorCtrl » avec une vue composite des ses éléments internes

Nous avons maintenant la possibilité de représenter les blocs fonction IEC 61131 avec une représentation quasi-identique à cette dernière. La Figure 7.3 montre comment il est ensuite possible de relier les ports de flux entre eux et d'obtenir une vue similaire à celle que l'on peut obtenir dans les diagrammes FBD de la norme IEC.

Les correspondances avec les ports de flux du bloc parent sont également illustrées. Le schéma ci-dessus est également extrait de l'article précédent qui contient le détail de la vue structurelle décrite.

7.3 Une Méthodologie Bidirectionnelle de Conception des Systèmes Automatisés (MBCSA)

Nous avons vu dans le chapitre sur l'approche objet et les différentes méthodes qui permettent de l'aborder que le principal risque dans le cadre d'analyse descendante est de concevoir des objets qui ne soient pas influencés par le contexte dans lequel ils sont spécifiés. C'est un risque car le concept même de l'objet veut qu'il soit défini de manière indépendante, offrant un mécanisme encapsulé dans son comportement et répondant à des sollicitations extérieures dont les demandeurs ne sont pas forcément définis lors de la réalisation de l'objet. Si ce n'est pas le cas, on peut avoir tendance à ajouter à l'objet des fonctionnalités supplémentaires pour qu'il puisse répondre à l'appel d'un autre objet avec lequel il est connecté dans une implémentation donnée, même si la fonctionnalité n'a pas trop de sens au sein de l'objet.

C'est la dérive qu'on observe souvent au niveau de la conception des objets. Pour aller vite dans leur définition, on a tendance à regrouper des comportements au sein d'un même objet pour qu'il réalise le maximum de fonctionnalités dans le cadre de ses interactions au coeur du système. On se retrouve ensuite avec un objet qui n'est pas réutilisable hors de contexte.

Partir des besoins utilisateurs au niveau d'un projet particulier est donc dangereux pour ce qui est de l'objectivité et de l'indépendance dans la définition des objets. Cependant, il faut bien partir de quelque chose quand rien n'est défini dans le système et les concepteurs, pour avoir une ligne de travail, se recentrent souvent sur une approche descendante partant des exigences utilisateur. De plus, la réutilisation des objets

passe également par une connexion d'éléments déjà existants au sein d'un élément de niveau structurel plus élevé. La spécification d'un projet global par la connexion d'éléments existants se fait donc également suivant une analyse descendante.

On observe donc un dilemme récurrent quant au choix du travail de spécification objet. Nous préconisons la séparation des spécifications en deux mouvements distincts à savoir:

- Une **Analyse Objet Montante** (**AOM**) ne se préoccupant pas du contexte de réalisation et se concentrant sur la spécification d'entités élémentaires.
- Une **Analyse Objet Descendante** (**AOD**) pour la réalisation de projet avec la réutilisation obligatoire d'éléments définis dans le premier cas.

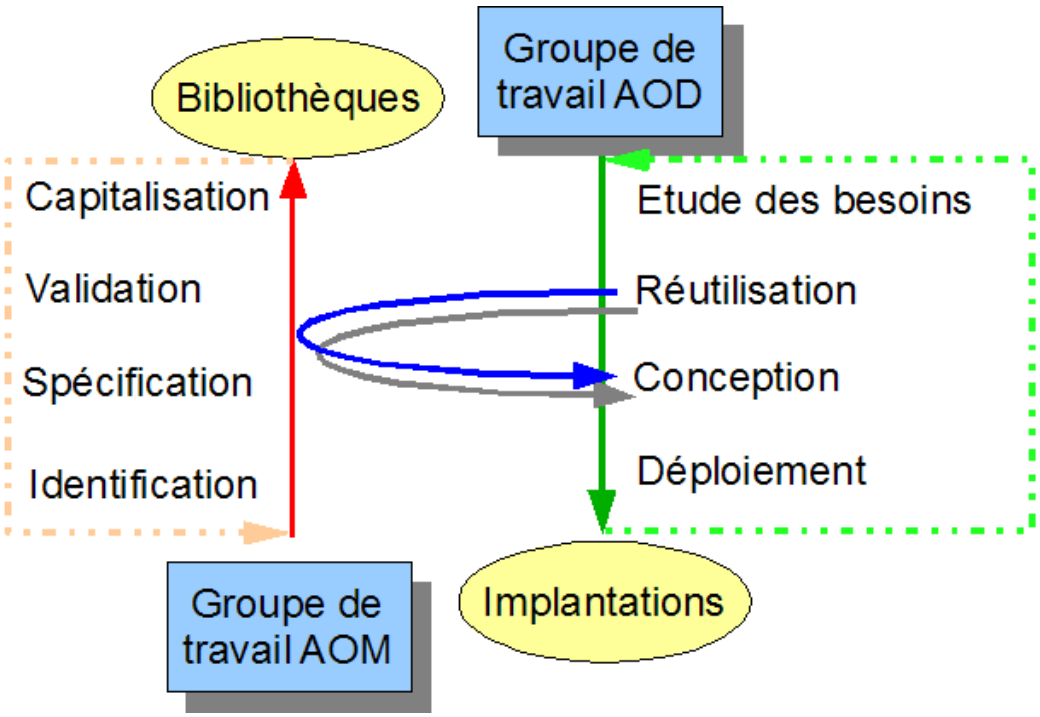


Figure 7.4.: Processus d'analyse bidirectionnel

On distingue donc le processus d'analyse qui va conduire à l'élaboration d'éléments individuels standardisés (**AOM**) et le processus d'analyse qui va être appliqué à chaque nouveau projet qu'il faudra réaliser et dont la principale tâche consistera à

intégrer et à connecter des éléments standards au sein du système (AOD).

Sur la Figure 7.4 nous exprimons cette vision graphiquement en détaillant les grandes étapes pour chacun des deux mouvements de spécification. L'**AOM** passe par les phases d'identification, de spécification détaillée, de validation et de test puis de capitalisation dans des bibliothèques. Les participants de l'**AOD** travaillant sur un projet donné partent d'une analyse des besoins, puis choisissent les composants à réutiliser, les connectent dans une phase de conception, puis les déploient vers les plates-formes cibles.

7.3.1 Description de l'AOM

Toute la difficulté de l'**AOM** est de savoir à quel degré de granularité il faut commencer l'analyse et ce que l'on doit analyser. Notre méthodologie étant destinée en premier lieu au domaine des systèmes de production contenant des éléments mécaniques, électriques et d'automatisme, nous nous plaçons dans ce contexte pour analyser ces deux questions.

7.3.1.1 Le niveau initial de granularité

« Il est relatif au métier de l'entreprise concerné et correspond aux entités non décomposables dont dispose une entreprise avant de les assembler dans le cadre des opérations industrielles qui la concernent ».

Cette description paraît vague au premier abord si elle est évaluée en dehors de tout contexte de production. Pourtant les personnes travaillant dans un métier donné peuvent identifier facilement les éléments de base qu'ils utilisent si on prend le temps de les questionner.

Pourquoi se limiter au niveau de décomposition des matières d'oeuvre en entrée ? Tout simplement dans l'optique d'un fonctionnement dans lequel c'est le fournisseur qui met à disposition la description de son matériel au sein d'un paquetage global de fourniture, spécifications standards et matérielles. Cette vision a l'avantage de laisser la description du matériel acheté à la charge des personnes qui en sont les spécialistes et donc d'assurer une bonne qualification du fonctionnement interne du

composant. De plus ces derniers n'étant pas nécessairement au courant du contexte dans lequel leurs éléments seront utilisés par la suite, on peut être optimiste quant à l'objectivité des spécifications fournies.

Actuellement, ces descriptions ne sont pas standardisées. On peut se retrouver avec un simple manuel, ou avoir la possibilité, par exemple en automatisme, de récupérer des fichiers de configuration pour l'intégration d'un matériel dans un réseau donné. Parfois il sera peut-être nécessaire de redéfinir certains aspects du matériel pour compléter ses spécifications. Cependant, la tendance actuelle est à l'établissement de règles standards de description, notamment dans l'automobile où le fournisseur d'un matériel devra apporter les plans mécaniques, le descriptif de fonctionnement, les plans électriques, la modélisation du programme dans la forme souhaitée par le client pour que ce dernier puisse les intégrer le plus rapidement possible dans ses propres processus de conception.

7.3.1.2 L'objet de l'analyse

Pour le constructeur de machines dans lesquelles cette thèse CIFRE s'est déroulée, les objets élémentaires sont par exemple des capteurs, des robots, des vérins, des systèmes de convoyage, des automates.

Pour un site de production, les objets seront d'un niveau de granularité plus important dans la perspective courante d'une sous-traitance de tous les modules physiques réalisant les différentes phases du flux de production. Il ne faut pas, au niveau de ce site de production, avoir à analyser la façon dont les modules sont réalisés et se contenter de les percevoir comme des boîtes noires opaques dont les seuls éléments connus sont les interfaces d'entrée et de sortie.

Le paragraphe suivant propose des pistes pour l'identification de ces objets élémentaires.

7.3.1.3 L'identification des briques élémentaires

Il est rare que les méthodes soient appliquées dans un contexte vide et nouveau. On a souvent la présence d'un existant au niveau de l'entreprise quand on

effectue une analyse de conception. Un moyen simple et efficace de répertorier des briques élémentaires est de regarder les commandes de matériel (nous ne prenons pas en compte à ce niveau et dans notre contexte d'étude la commande de logiciels) puis de reprendre la liste de tous ces éléments telle quelle, sans aucun critère de sélection (pupitre de commande, colonne lumineuse, armoire, cellule, vérin).

La phase suivante de spécification peut commencer dés la détection du premier élément.

7.3.1.4 La spécification multi-facettes des briques élémentaires

Il s'agit ici d'élargir le concept d'objet mécatronique que l'on peut trouver dans les travaux de M. Bonfè [Bonfè 2000], [Bonfè 2001] en se plaçant dans les systèmes de production automatisés. C'est ici que nous utilisons le concept de composant multi-facettes pour exprimer le fait que l'analyse de l'élément va s'effectuer selon différents points de vue. Nous utilisons la modélisation **UML** pour la spécification et, afin de structurer notre description, nous caractérisons un élément dans un premier niveau de modèle par un paquetage **UML**.

Chaque facette sera décrite par un ou plusieurs modèles **UML** que nous répertorions plus loin et qui constitueront au fur et à mesure la modélisation complète du composant de façon à pouvoir être insérés par la suite dans la réalisation à proprement dite des projets. On retrouve une constance au niveau de l'enchaînement des descriptions de chaque facette :

- On commence par considérer l'élément dans le contexte de la facette avec un regard extérieur au composant, sans s'intéresser à l'implémentation interne, en se demandant de quoi ce composant a-t-il besoin en entrée pour fonctionner puis ce qu'il va pouvoir fournir à son environnement après traitement de sa fonction. On retrouve cette vision sous le très répandu concept de « *boîte noire* ».

- L'étape suivante consiste à éclater la vue précédente pour identifier les éléments encapsulés dont la collaboration va permettre de réaliser le comportement interne du composant considéré. On utilise le terme « *boîte blanche* » pour

caractériser cette vue. Cette seconde identification n'est normalement pas nécessaire si on est en possession d'une brique déjà élémentaire. Si cette brique contient une structure interne dont la connaissance n'est pas nécessaire pour le travail de conception qui s'effectuera ultérieurement en réutilisant la brique considérée, on ne descendra pas au niveau de description inférieur.

– La dernière vue se concentrera sur une analyse de la dynamique du composant mais dans le cadre de la facette étudiée.

Dans le cadre des systèmes automatisés, nous avons identifié cinq facettes à passer en revue pour chaque élément. Il n'est pas nécessaire de décrire chaque facette, certains éléments ne sont pas forcément concernés par tous les aspects que l'on se propose d'étudier. Il faut cependant que la facette existe, même si elle est vide, pour maintenir la cohérence structurelle de l'assemblage par la suite.

Les paragraphes suivants présentent, pour chaque facette, ce que nous préconisons d'utiliser au niveau de la description basée sur **UML** en précisant les diagrammes et les concepts à choisir.

7.3.1.4.1 Facette de Dialogue Homme-Machine

Typiquement il s'agit de décrire les services fournis ou requis par l'élément en terme d'interactions avec un utilisateur. Nous représentons d'abord l'élément en utilisant un diagramme de composant et en ajoutant en premier lieu une interface requise puis une interface fournie toutes les deux vides.

A ce niveau, il est intéressant d'interroger une personne du métier en lui demandant notamment ce qu'il peut avoir à faire directement avec l'élément même si la taille de l'élément est très petite. Par exemple pour le cas d'un capteur, on peut souhaiter lire l'état de ce dernier, le visualiser en 2 ou 3 dimensions, éventuellement agir dessus, lancer une procédure de test de bon fonctionnement, changer sa sensibilité. De même on pourra imaginer que le capteur nécessite une procédure de réglage initiale, ou de dépannage lorsqu'un problème est détecté.

Pour savoir s'il faut tout décrire dans la même interface ou utiliser des interfaces différentes nous considérons l'état de l'élément lorsque les services sont

utilisés. Tous les services, pouvant être utilisés dans le même ensemble d'états, peuvent être réunis dans la même interface. On pourra ainsi placer dans une même interface des services sollicités pendant le fonctionnement normal du capteur, et dans une autre ceux pouvant être sollicités lorsqu'il est hors service. Si l'on prend le cas des interventions de maintenance périodiques, on définira une interface requise de demande de maintenance au niveau du composant.

Ces interfaces fournies seront directement reliées à des ports de type protocolaire sur la classe. Ceci nous permet de décrire l'utilisation de l'interface dans une machine à états de protocole. L'objectif est de décrire précisément l'enchaînement des opérations internes du composant avec les états successifs de ce dernier, provoqués par la sollicitation d'un service.

En complément, pour une visualisation des interactions entre objets internes de la classe, on utilisera un diagramme de séquence **UML**.

La caractérisation des interfaces devra également permettre l'identification des attributs et des méthodes internes au composant qui permettront de réaliser correctement les services. Par exemple pour le paramétrage, on peut se servir d'une boîte de dialogue avec des champs de saisie et des boutons de validation. La visualisation de l'état peut se faire par l'affichage simple d'un champ avec une valeur. Les attributs liés à des objets graphiques, bien que d'un niveau de granularité inférieur, ne sont pas à définir ici. On considère que l'on a accès à une librairie d'objets graphiques nous permettant de représenter ce que l'on veut et on se limite à la caractérisation de l'élément parent.

7.3.1.4.2 Facette logique de fonctionnement

Pour un système automatisé, on représente cette partie en se basant sur un langage de la norme **IEC 61131**. Pour rester conforme à l'approche objet, il faut ici imaginer que le déploiement de cette facette correspondra à une implémentation en tant que bloc fonction pour pouvoir être compilé dans un environnement d'automatisme.

Nous choisissons d'utiliser l'extension **SysML,** à ce niveau, pour représenter à l'aide d'un diagramme de définition de bloc, la vue externe sous forme de boîte noire de l'élément considéré. Il s'agit d'identifier les variables dont il va avoir besoin pour

fonctionner correctement à l'aide « *flow ports* » avec le préfixe « *in* » ou « *inout* » (si la variable que l'on traite est transmise en sortie également) et les données qu'il va mettre à jour et à disposition de son environnement extérieur avec le préfixe « *out* ». La symbolique **SysML** pour ce type de variable a été présentée précédemment dans la partie introduisant **SysML**. Pour que le bloc puisse accomplir sa tâche, il faut un élément déclencheur, correspondant au flux de contrôle de l'application, que nous modélisons comme un service de déclenchement au niveau d'une interface fournie. De même, une fois les opérations internes terminées, le composant va à son tour renvoyer le flux de contrôle en sortie, ce que nous modélisons par la présence d'une interface requise.

Pour la vue structurelle interne de ce composant, nous utilisons un diagramme interne de bloc de la même façon que sur la Figure 7.2, en reliant les « *flow ports* » des éléments internes entre eux ou avec ceux du composant parent.

Une fois ces deux angles de considération (vue externe/vue interne) décrits, un diagramme d'activité permet de détailler comment sont utilisées les données d'entrée ou comment sont implémentées celles de sortie par le composant. Ces données sont modélisées en tant que paramètres d'entrées/sortie au niveau de l'activité globale comme « *e_delay* », « *e_S* », « *s_SU* », « *s_SD* » sur la Figure 7.5. On retrouve la description dynamique de la décomposition structurelle de la Figure 7.2

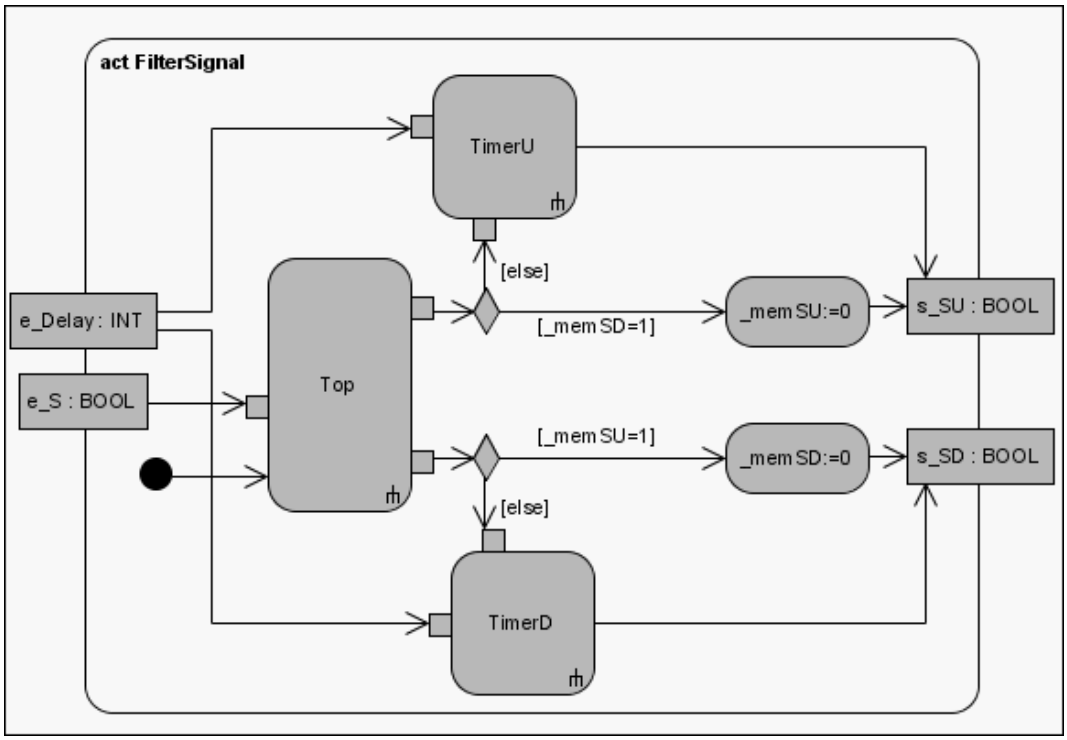


Figure 7.5.: Diagramme d'activité d'un bloc SysML, avec la présentation des paramètres en entrée/sortie

7.3.1.4.3 Facette de description physique ou mécanique

De même que pour la facette précédente, la première étape est la représentation sous forme d'un bloc **SysML**, avec deux interfaces vides, une requise et une fournie, dans un diagramme de composant interne au paquetage parent. Nous préconisons ensuite l'utilisation d'un diagramme de structure composite pour la description des éléments physiques qui composent l'élément sur lequel nous nous penchons.

L'article [Björkander 2003] co-écrit par C. Kobryn, l'un des pères d'**UML**, présente un exemple intéressant de la vue composite simplifiée de la structure physique interne d'un véhicule. L'utilisation d'**OCL** va également permettre de détailler le diagramme en terme de contraintes physiques dans les relations entre sous-éléments

mécaniques, par exemple un couple, une force de frottement, un débit maximal.

S'agissant de la vue globale du composant, on fera apparaître au niveau des interfaces les besoins physiques en terme d'alimentation énergétique (pneumatique, électrique, hydraulique) ou bien de forces physiques par exemple un couple, une pression, nécessaires à l'élément pour fonctionner correctement. Ces interfaces ne seront pas liées à un port protocolaire.

On utilise ensuite un diagramme d'états de comportement au niveau du composant pour représenter les différents états physiques de l'élément et la façon dont on passe de l'un à l'autre, avec la sollicitation éventuelle de ses sous-éléments.

7.3.1.4.4 Facette de description électrique

Lors de la conception d'un système automatisé, on passe par une phase de réalisation des schémas électriques, c'est à dire que l'on représente avec une symbolique graphique standardisée le câblage électrique des différents éléments en les connectant ensemble ou en les reliant à des sources extérieures.

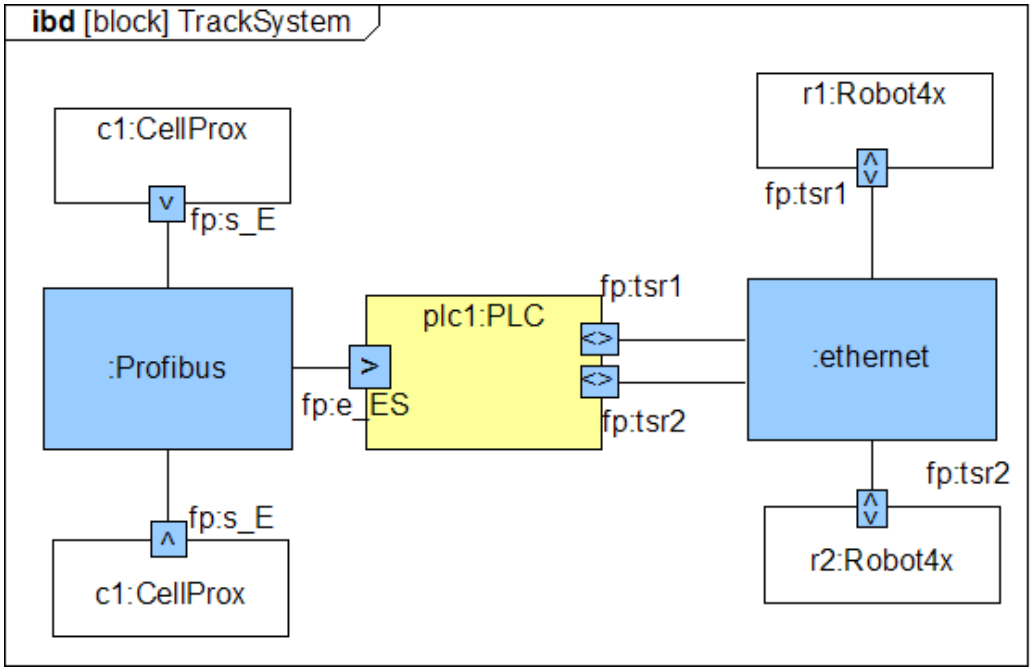


Figure 7.6.: Diagramme interne de bloc, vue du réseau de communication

Pour un élément donné, spécifier un tel schéma revient à connecter ses éléments internes sur une représentation de réseau. Des sous-éléments qui ne sont pas reliés mécaniquement peuvent être reliés au niveau de cette représentation, par exemple s'ils sont positionnés sur un même bus de terrain, il est donc important de représenter cet aspect au niveau d'un élément composant un système automatisé.

Pour représenter cette facette nous préconisons l'utilisation d'un diagramme interne de bloc (*ibd*) comme sur la Figure 7.6 avec la modélisation du médium de communication sous forme d'instance de classe globale et la connexion des éléments communiquant via ce réseau, par exemple les robots r1 et r2 sont reliés à l'automate en passant par un réseau Ethernet.

7.3.1.4.5 **Facette MES**

Le MES (*Manufacturing Execution System*) englobe les considérations se situant entre la production elle-même, avec ses différents éléments d'automatisme, de supervision, et les parties supérieures chargées de la planification comme l'**ERP** (*Enterprise Ressource Planning*) ou la **GPAO** (*Gestion de Production Assistée par Ordinateur*). Son rôle est d'assurer la traçabilité des systèmes de production et de leur fournir les ordres de fabrication.

L'objectif de cette facette est donc de se placer au niveau de l'élément décrit et de réfléchir sur des considérations au niveau du **MES**. Dans le cas où il est impossible de connaître les raisons d'un accès externe à une valeur, on se limite à la description d'une interface requise d'accès à des services d'écriture ou de lecture offerts par un medium de communication ou de stockage que l'on ne connaît pas à ce moment (stockage **XML**, requêtes **SQL**...). On représente cette interface sur un diagramme de composant de façon classique avec les services requis et fournis présentés dans des interfaces.

7.3.1.5 La validation des briques élémentaires

La validation de ces briques élémentaires passe par une phase de test sur une plate-forme adéquate. Il est probable que dans le futur, nous aurons la possibilité de tester directement le comportement des modèles au niveau de l'outil de modélisation (ce qui

n'existe pas pour le moment avec **SysML**). Le principe est de simuler l'état des entrées, de les faire évoluer et de visualiser l'impact sur les sorties et la façon dont les éléments internes se comportent. Un élément sera donc considéré comme valide à l'issue de ce test. Lors de sa réutilisation ultérieure au sein d'un système plus global, l'utilisateur aura ainsi des garanties sur le fonctionnement de l'élément qu'il insère dans son système.

Souvent les fournisseurs de matériel mettent à disposition le comportement des éléments qu'ils vendent, par exemple sous forme de chronogrammes. L'objectif de cette partie est similaire, on cherche à fournir un manuel de comportement dynamique validé et dont on se porte garant, indépendamment d'un contexte d'utilisation.

7.3.1.6 La capitalisation

Une fois les différentes facettes spécifiées et le comportement de l'élément validé, ce dernier peut être stocké dans la bibliothèque d'éléments disponibles pour les personnes en charge de l'**AOD**.

L'élément est stocké sous forme de package mais ce sont bien les composants décrits dans les différentes facettes qui seront utilisables dans les modèles conçus lors de l'**AOD**.

7.3.2 Description de l'AOD

Le contexte de l'**AOD** est différent puisqu'il correspond à la réalisation d'un projet de conception de systèmes beaucoup plus complexes, dont les éléments ne sont pas identifiables aussi facilement que précédemment.

Il faut prendre en compte des contraintes supplémentaires qui vont guider les choix au niveau de ces éléments pour pouvoir réaliser les fonctions attendues.

7.3.2.1 Analyse des besoins du système

Durant cette phase on retrouve une analyse classique d'identification des besoins auxquels le système devra répondre. L'utilisation d'un diagramme de cas d'utilisation permet d'initier ce travail avec la représentation des acteurs du système et des

interactions attendues avec le système comme sur la Figure 6.18.

Une fois les cas d'utilisation identifiés, l'emploi de diagrammes d'activité permet de détailler les différentes opérations réalisées par le système pour les différents acteurs. On décrit les scénarii et l'enchaînement des actions qui conduira à la réalisation du service sous une forme fonctionnelle non détaillée. La Figure 7.7 montre un exemple de scénario pour un cas d'utilisation permettant à un utilisateur de réaliser un programme de palettisation.

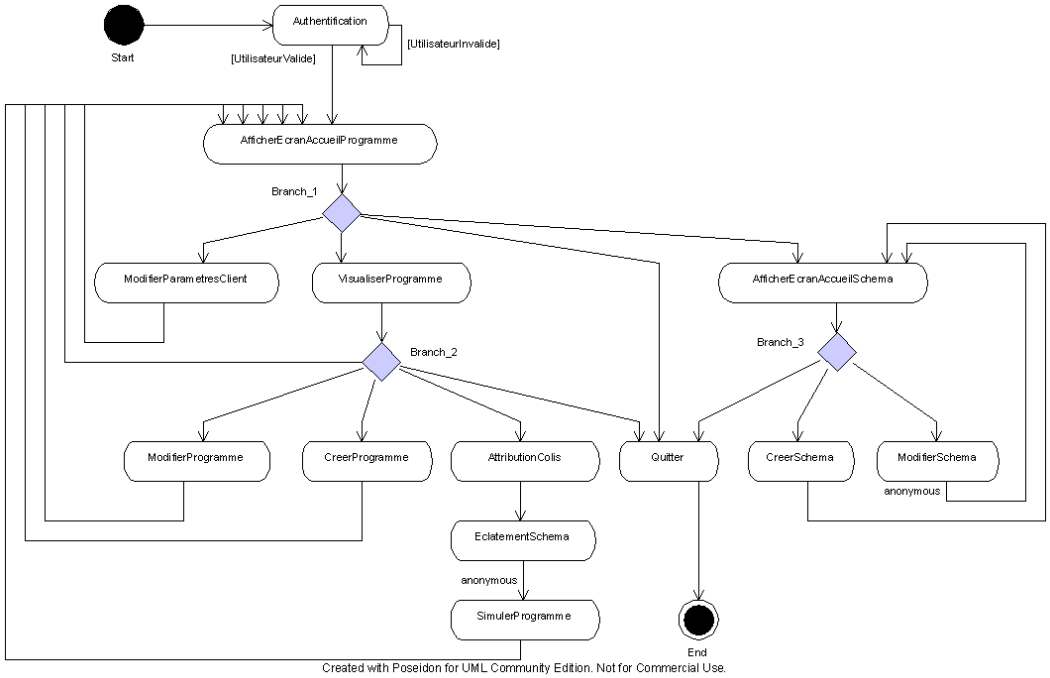


Figure 7.7.: Diagramme d'activité, scénario détaillant un cas d'utilisation

7.3.2.2 Choix des éléments de composition

A ce stade du travail, l'analyse fonctionnelle des besoins et la description préliminaire des différents scénarii de fonctionnement ont été complétées. La conception détaillée peut démarrer avec dans un premier temps la consultation des spécialistes de développement concernés par les facettes identifiées précédemment.

En analysant avec les concepteurs les actions présentes dans les scénarii

fonctionnels précédents, il va être possible d'établir les listes d'éléments que chaque opération devra avoir à sa disposition pour s'effectuer correctement.

Trois cas se présentent alors pour chaque élément requis :

- L'élément est une brique élémentaire disponible dans l'une des bibliothèques établies par les personnes chargées de l'**AOM**.
- L'élément est une brique élémentaire mais n'est pas disponible; une requête peut alors être faite au niveau de l'**AOM** pour la spécification d'un nouvel objet.
- L'élément est une composition d'éléments existants, il devra être détaillé dans la phase suivante.

7.3.2.3 Conception par composition

Cette phase constitue la raison d'être de l'approche objet; les éléments ont été identifiés, il s'agit de les assembler entre eux dans le contexte d'un scénario et d'une facette pour obtenir le comportement global attendu.

Lorsque la connexion correspond à une sollicitation de services entre deux composants, ou une liaison entre deux ports de flux de données dans le cas des « *block* » **SysML**, il suffit de représenter le lien dans le diagramme de description interne correspondant à la facette décrite.

Dans le cas d'une interaction plus complexe entre deux composants, il est nécessaire de définir un nouvel objet intermédiaire. Les collaborations sont des éléments de modélisation qui permettent de spécifier ces relations. De plus leur utilisation permet également de capitaliser des interactions récurrentes sous forme de classes.

Dans la phase précédente, nous avons établi les éléments qui permettront de réaliser les fonctions attendues du système et ceci dans les différentes facettes d'étude. Il peut être nécessaire d'insérer de nouveaux éléments si les interfaces requises des composants déjà identifiés n'ont pas toutes la possibilité d'être reliées, ce qui permet de compléter les diagrammes contenant les compositions d'éléments.

On doit finalement obtenir pour chaque facette du système un diagramme de

structure et les diagrammes de comportement associés, pour le premier niveau de composition. Les éléments compris dans ces diagrammes sont eux-mêmes décomposables à leur tour (Figure 7.8 sans la facette MES) jusqu'à aboutir aux briques élémentaires non décomposables.

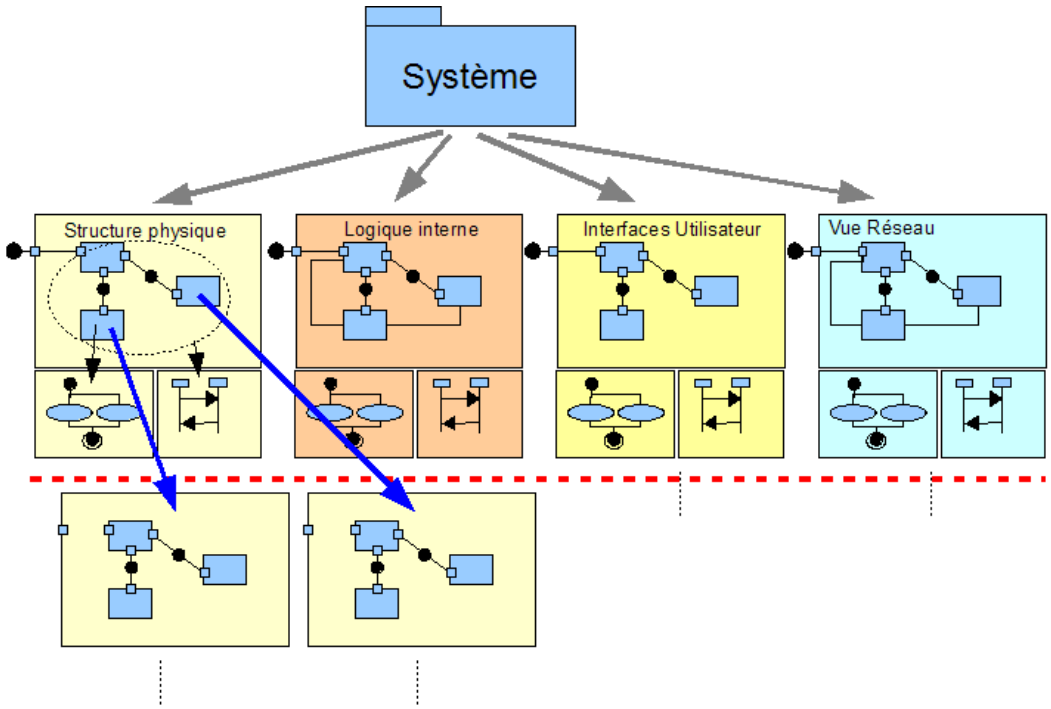


Figure 7.8.: Niveaux de composition par facettes du système

7.3.2.4 Déploiement des modèles

Selon l'approche **MDA**, il est important de considérer les éléments indépendamment d'un contexte de déploiement donné. Bien que nous ayons représenté la définition du système en séparant les domaines de spécification de façon bien précise, il ne faut pas utiliser des éléments de description liés à une technologie donnée. Ce choix sera fait au dernier moment, l'objectif étant de générer automatiquement une grande partie des spécifications.

On passera donc par des langages standards intermédiaires, reconnus pour un domaine donné.

7.3.2.4.1 Interfaces utilisateurs

Elles peuvent être réalisées dans un environnement fourni avec le matériel de supervision, des « *mapping* » spécifiques sont à réaliser dans ce cas. Nous nous intéressons plutôt aux langages informatiques de programmation traditionnels et pour lesquels il existe déjà de nombreux modules de génération de code automatique utilisables à partir des outils de modélisation du marché.

```
typedef struct tagNATTRIBCOL
{
    DWORD    dwXmin;
    DWORD    dwXmax;
    DWORD    dwYmin;
    DWORD    dwYmax;
    long     lTeta;
    WORD     wRobot;
    WORD     wVoie;
    WORD     wGroupe;
    WORD     wTranche;
}  NATTRIBCOL;
typedef struct tagNATTRIBSCH
{
    DWORD    dwColisCount;
    WORD     wXmax;
    WORD     wYmax;
    [size_is(dwColisCount)] NATTRIBCOL* pColis;
}  NATTRIBSCH;
typedef struct tagNPROG
{
    DWORD    dwSchemaCount;
    [size_is(dwSchemaCount)] NATTRIBSCH*    pSchema;
}  NPROG;
typedef struct tagNCAMP
{
    DWORD    dwProgCount;
    [size_is(dwProgCount)] NPROG*   pProg;
}  NCAMP;
[
    object,
    uuid(C2BADB25-274E-40CE-90D9-55E2BF589724),

    helpstring("ICoAttribution Interface"),
    pointer_default(unique)
]
interface ICoAttribution : IUnknown
{
    [helpstring("method AttribuerCampagne")] HRESULT AttribuerCampagne(
    [in] WORD wIdCampagne,
    [in] WORD wIdImplantation,
    [in] WORD wIdProduit,
    [out] NCAMP* pCampagne,
    [out] WORD* pError);
};
```

Figure 7.9.: Exemple de génération de description d'interface pour un composant de la partie Dialogue Homme Machine

Pour les interfaces, nous préconisons de passer par la génération d'une description avec le format standard de l'**OMG**, l'**IDL** (*Interface Definition Language*),

utilisé initialement dans le cadre de **CORBA** mais également intégré au niveau du langage de définition des interfaces **COM** (*Component Object model*), technologie choisie dans notre cas pour la réalisation des composants de l'interface Homme-Machine et dont un exemple est donné sur la Figure 7.9.

Pour notre cas d'étude, nous avons utilisé la technologie **COM** et les **MFC** (*Microsoft Foundation Classes*). Deux modèles intermédiaires ont donc été réalisés, un premier intégrant les stéréotypes standards **C++** ainsi que les collections de la **STL** (*Standard Template Library*), puis un second effectuant un choix plus précis au niveau de la programmation des interfaces graphiques avec les **MFC**. La génération de ce modèle spécifique a permis d'obtenir les squelettes de classes de l'application pour ces technologies. La Figure 7.10 reprend ces étapes de spécialisation.

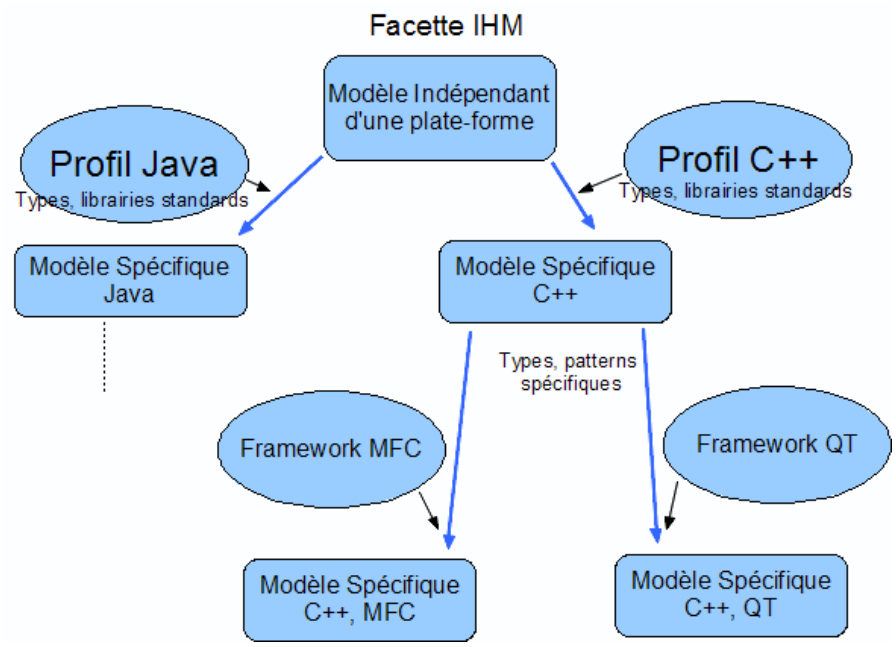


Figure 7.10.: Spécialisation des modèles décrivant la facette IHM d'un composant multi-facette

7.3.2.4.2 Logique Interne

Cette facette contient la description du comportement de l'élément intégré dans le système automatisé. L'objectif du déploiement à ce niveau est de pouvoir passer

des représentations **SysML** décrites précédemment au code de fonctionnement utilisé sur la machine finale, le plus souvent un automate. Cette phase est décrite plus précisément dans la partie 7.4.

7.3.2.4.3 Les autres facettes

Au niveau de la représentation de la structure mécanique, l'objectif de la phase de déploiement serait d'obtenir par génération des fichiers exploitables pour la réalisation mécanique du système au niveau des outils **CAO** (*Conception Assistée par Ordinateur*) de représentation 3D du marché. Il existe dans ce domaine de nombreux standards d'échange (IGES, STEP, 3DS, MD3, X, XGL, etc...), qui pourraient constituer une cible technologique pour la génération automatique des modèles de structure physique.

L'aspect réseau de l'**AOD** permet quant à lui de définir les profils qui seront à utiliser pour deux facettes: **IHM** et logique de fonctionnement. On visualise en effet les plates-formes technologiques choisies lors de la conception du système, les protocoles utilisés pour communiquer entre elles, et la répartition des éléments modélisés entre les différentes plates-formes. Pour caractériser ces plates-formes, on parle de conteneur. Un conteneur va accueillir un certain nombre de composants en fournissant un ensemble de services non-applicatifs dont ils ont besoin pour communiquer ou fonctionner. Cette couche de services appelée *intergiciel* (plus communément *middleware*) permet aux composants de faire l'abstraction d'un certain nombre de fonctionnalités liées à des services qui sortent du contexte particulier d'un composant. On retrouve par exemple dans notre contexte les services d'accès à une base de données, les services d'accès à un serveur **OPC** ainsi que tous les services de communication liés à un protocole donné.

Cette facette du système est donc essentielle et représente un moyen clair de cibler les plates-formes de destination ainsi que les composants qui sont déployés ou non sur celles-ci, afin de choisir les différents profils à utiliser lors de la génération des éléments de code.

7.4 Le déploiement automate

7.4.1 Le formalisme PLCopen

7.4.1.1 Présentation

Si l'on regarde les outils de modélisation **UML**, il est possible d'échanger très facilement des modèles entre outils grâce à l'existence du standard **XMI** et à la standardisation de la représentation de la sémantique **UML** en **XML**. Une branche de l'activité du comité **PLCopen** a produit et lancé un groupe de travail pour effectuer la mise en place d'une correspondance similaire entre langages **IEC 61131** et un format de représentation standard basé sur la sémantique **XML**.

Ces efforts ont conduit à l'élaboration d'un document [PLCOPEN 2004], produit par le comité technique 6 (*PLCopen Technical Commitee 6,* ***TC6***). Il est important de noter l'implication des industriels concernés par la finalité de ce travail de standardisation, avec notamment les fournisseurs d'automates dont les principaux acteurs du marché sont à l'heure actuelle *Siemens*, *Schneider Automation*, *Rockwell Automation, B&R* et *Beckhoff*.

Si l'on regarde les produits actuels du marché, avec par exemple *STEP7* de *Siemens* et *Unity Pro* de *Schneider Automation*, il est encore impossible de passer d'un environnement à un autre en utilisant des exportations ou des importations **XML** dans un format d'échange standard même si avec *Unity Pro,* Schneider a ouvert son ancien format de stockage pour le proposer en **XML** avec un fichier de structure personnalisé.

De plus, les fournisseurs de moindre envergure, souhaitant percer sur le marché des environnements de programmation n'hésitent plus à ouvrir leurs formats d'échanges et mettent en avant pour les clients cette caractéristique; l'entreprise *TNI Software* en particulier et son produit *Control-build 4.1* proposent l'exportation des programmes réalisés sous leur plate-forme de façon totalement conforme aux spécifications du consortium PLCopen.

On peut donc être malgré tout optimiste quant à l'avenir de l'ouverture des

standards de stockage pour les programmes automates et cela rend d'autant plus pertinente l'idée d'une conception de programme indépendante d'une plate-forme donnée. Le code généré serait ensuite ouvert dans l'environnement de programmation correspondant au matériel ciblé puis compilé de façon optimale pour ce même matériel.

7.4.1.2 Les éléments de la représentation

La structure de représentation utilisée pour **XML** est détaillée dans les paragraphes qui suivent.

7.4.1.2.1 **Structure du projet**

Elle décrit les caractéristiques générales de l'ensemble du programme. Dans le champ « *fileheader* » on retrouve les informations concernant l'entreprise, la description et la version du projet, la date d'édition. Un second champ « *contentheader* »contient des informations plus détaillées sur le projet en lui-même avec notamment la description du système de coordonnées exporté pour les langages graphiques (**SFC**, **LD**, **FBD**).

7.4.1.2.2 **Définition des types**

Dans cette partie on définit, dans un premier temps, les types de données qui ne sont pas des types élémentaires dans le champ « *dataTypes* » (BOOL, BYTE, WORD, DWORD, LWORD, SINT, INT, DINT, LINT, USINT, UINT, UDINT, ULINT, REAL, LREAL, TIME, DATE, DT, TOD, STRING, WSTRING), par exemple des structures contenant un ensemble de types de données, des tableaux à plusieurs dimensions.

Ensuite, on peut détailler toutes les unités d'organisation de programme (**POUs**) selon le schéma **XML** de la Figure 7.11 pour le champ « *pou* ». Les unités d'organisation vont par exemple contenir la description des blocs fonction que nous voulons obtenir à partir des blocs **SysML**, la description des connexions entre ces blocs et l'implémentation interne de ces blocs comme nous le verrons dans la partie 7.4.2.

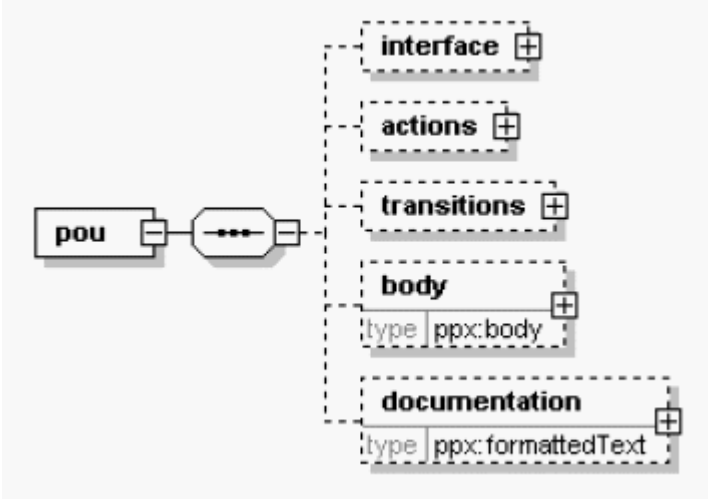


Figure 7.11.: Schéma de structure PLCopen (issu des spécifications) pour la description des Unités d'Organisation de Programme

Au niveau du schéma **XML** de l'unité d'organisation de programme, on retrouve des informations sur les variables utilisées dans la balise « *interface* », définies ou mises à jour par la portion de code correspondante. On liste ainsi les variables définies localement, les variables globales ou externes, les variables d'entrée, de sortie ou d'entrée/ sortie, les variables temporaires et les variables localisées (liées à des adresses physiques).

Les parties « *actions* » et « *transitions* » regroupent une liste, pouvant être vide, de plusieurs références à des sections de codes décrites dans le champ « *body* ». Elles référencent notamment les étapes et transitions constituant les **POUs** décrites dans le langage **SFC**.

La description contenue dans le champ « *body* » décrit directement le programme constituant la **POU** avec un formalisme directement dépendant du type de langage qui a été choisi pour l'implémentation. La Figure 7.12 montre le schéma de structure **XML** pour cette partie avec les sous-balises correspondant aux différents langages. La description des langages graphiques est particulièrement détaillée en terme de positionnement et de symbolique pour que les diagrammes soient parfaitement retranscrits graphiquement entre outils. Pour les langages dits textuels, comme le langage LIST ou Structuré, le code est écrit de façon brute, sans balises particulières.

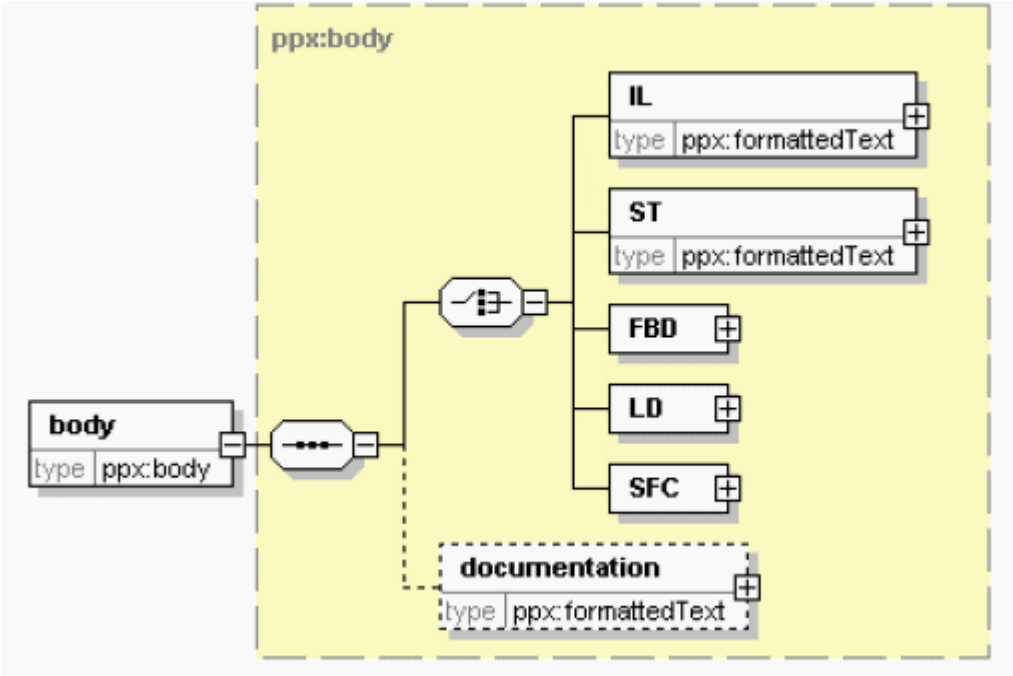


Figure 7.12.: Schéma de structure PLCopen (issu des spécifications) pour la description du corps de programme d'une Unité d'Organisation de Programme

7.4.1.2.3 Définitions des instances

Cette partie permet notamment d'associer les **POUs** précédemment définies à des ressources ou à des tâches. Le champ « *configuration* » contient ces correspondances au niveau du code **XML**. La notion de tâche notamment permet de regrouper les sections de codes contenues dans les **POUs** pour qu'elles s'exécutent selon un rythme donné (cyclique ou périodique) ou lors du déclenchement d'évènements bien précis (*triggered tasks*).

7.4.2 La génération de code

Pour le déploiement automate nous définissons le processus de génération de code visible sur la Figure 7.13 avec une première étape passant par une représentation conforme aux recommandations PLCopen pour l'échange de fichiers de code automate,

puis le choix de certains langages de la norme **IEC 61131** en fonction de nos besoins pour un traitement final au niveau des fichiers **XML** afin de se conformer aux spécifications de la plate-forme ou de l'environnement de programmation ciblé.

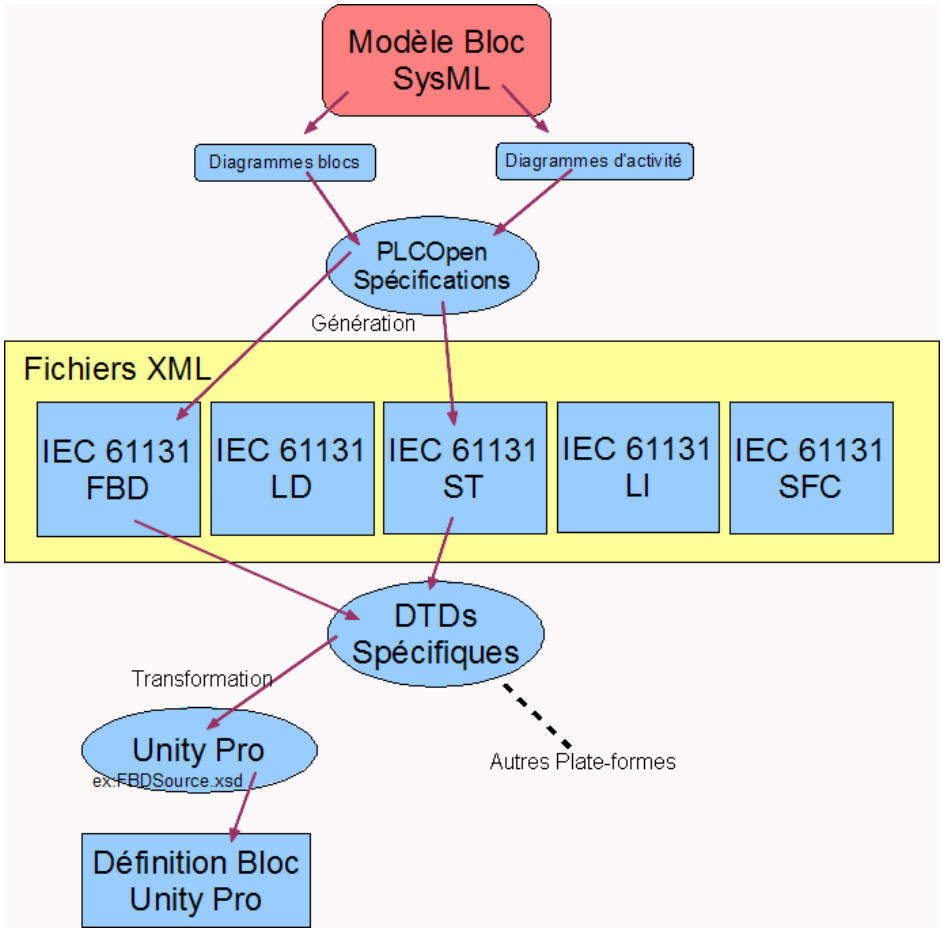


Figure 7.13.: Processus de génération de l'implémentation de la logique automate

7.4.2.1 Obtention d'un premier fichier de spécification XML

Si l'on reprend les facettes de logique de fonctionnement représentées en **SysML**, nous avons des diagrammes représentant des blocs sous deux aspects, structurel dans les diagrammes de *définition* ou *internes* de blocs, et dynamique dans les

diagrammes d'activité associés aux blocs.

Nous choisissons de passer par les spécifications de **PLCopen** pour la définition de la logique de fonctionnement lors de la génération de code à partir des modèles **SysML**. Les spécifications **TC6** ne préconisent pas le choix d'un langage particulier pour la représentation du programme, l'objectif étant de pouvoir échanger des éléments de code dans n'importe quel langage de la norme **IEC 6113**1 entre différents outils. Nous avons donc fait le choix de deux langages de la norme pour les deux principaux aspects de notre représentation en **SysML**.

Pour la partie structurelle du système, vue sous la forme d'une connexion de sous-éléments multiples comme sur la Figure 7.13, nous décidons de garder l'aspect graphique de la représentation **SysML** en effectuant une projection vers la symbolique utilisée dans les diagrammes **FBD** de la norme **IEC 61131**.

Pour la description dynamique interne des blocs, nous optons pour le langage structuré (**ST**) qui permet de décrire des logiques de programme complexes de façon beaucoup plus pratique que dans les autres langages. De plus, le langage structuré permet de garder une façon de programmer très proche de celle utilisée dans les langages informatiques traditionnels, notamment pour la réutilisation d'objets instanciés, ce qui permet de faire facilement le lien avec les diagrammes comportementaux d'**UML;** l'inconvénient étant que les spécifications PLCopen pour ce langage ne proposent pas de structure **XML** pour décomposer la description de cette logique, ce qui aurait permis de relier les éléments **UML** du diagramme d'activité à certaines balises. Nous pensons cependant que cette description plus détaillée sera possible à l'avenir.

7.4.2.1.1 Définition statique du bloc

Le langage **FBD** est basée sur une représentation graphique, l'emplacement des blocs sur le diagramme doit donc être précisé lors du transfert des informations du diagramme vers d'autres destinations. La Figure 7.14 montre la structure de représentation **XML** des blocs présents dans les diagrammes **FBD** avec le premier champ « *position* ». Cette description du positionnement est issu directement de celui des blocs **SysML** sur les diagrammes *bdd* et *ibd*.

En fonction du préfixe définissant les *flow ports* rattachés au bloc SysML, le contenu des balises « *inputVariables* », « *inoutVariables* » et « *outputVariables* » est renseigné.

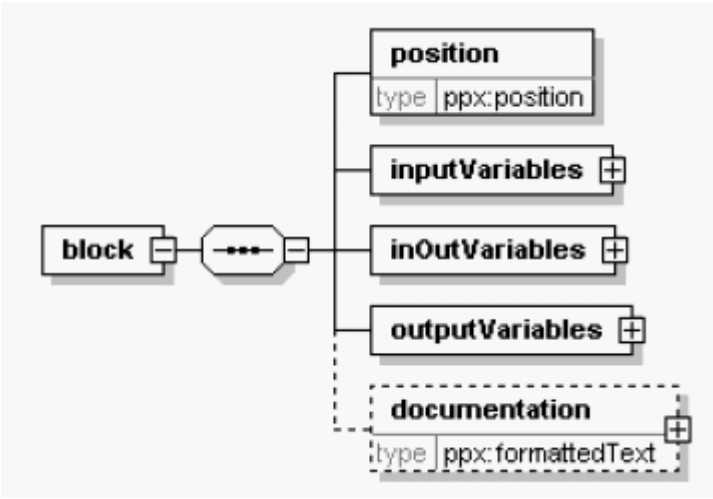


Figure 7.14.: Schéma de structure PLCopen (issu des spécifications) pour la description d'un bloc fonction au sein d'un diagramme FBD

7.4.2.1.2 Définition dynamique du bloc

Nous avons retenu le langage Structuré (**ST**) pour l'implémentation du comportement interne des blocs. Ce langage permet en effet la représentation de logiques complexes sous une forme proche de celles qui sont modélisées dans les diagrammes d'activité (contrairement au langage **LIST**).

7.4.2.1.3 Les fichiers de génération

En reprenant l'exemple de la Figure 7.3, on peut ainsi obtenir un fichier

```
<?xml version="1.0" encoding="utf-8"?>
<project xmlns="http://www.plcopen.org/xml/tc6.xsd"
xmlns:xhtml="http://www.w3.org/1999/xhtml"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://www.plcopen.org/xml/tc6.xsd
http://www.plcopen.org/xml/tc6.xsd">
  <fileHeader companyName="newtec"
```

```
creationDateTime="2006-09-15T10:13:00"
productName="NoNameNow"
productVersion="1"/>
<contentHeader name="SignalFilter">
  <coordinateInfo>
    <fbd>
      <scaling x="16" y="16"/>
    </fbd>
    <ld>
      <scaling x="16" y="16"/>
    </ld>
    <sfc>
      <scaling x="16" y="16"/>
    </sfc>
  </coordinateInfo>
</contentHeader>
<types>
  <dataTypes/>
  <pous>
    <pou name="SignalFilter" pouType="functionBlock">
      <interface>
        <localVars>
          <variable name="_memSU">
            <type>
              <BOOL/>
            </type>
          </variable>
          <variable name="_memSD">
            <type>
              <BOOL/>
            </type>
          </variable>
        </localVars>
        <inputVars>
          <variable name="e_S">
            <type>
              <BOOL/>
            </type>
          </variable>
          <variable name="e_Delay">
            <type>
              <INT/>
            </type>
          </variable>
```

```
      </inputVars>
      <outputVars>
       <variable name="s_SU">
        <type>
         <BOOL/>
        </type>
       </variable>
       <variable name="s_SD">
        <type>
         <BOOL/>
        </type>
       </variable>
       <variable name="s_S">
        <type>
         <BOOL/>
        </type>
       </variable>
      </outputVars>
     </interface>
     <body>
      <ST>
       top(e_S:=e_S);
       if top.s_SU then
        _memSD:=0;
        _memSU:=1;
       else if top.s_SD then
        _memSD:=1;
        _memSU:=0;
       end_if;

       TimerD (PT:=e_Delay, IN:=_memSD);
       TimerU (PT:=e_Delay, IN:=_memSU);

       s_SU:=TimerU.Q;
       s_SD:=TimerD.Q;
      </ST>
     </body>
    </pou>
   </pous>
  </types>
  <instances>
   <configurations/>
  </instances>
</project>
```

Avec l'utilisation d'un convertisseur **XML**, on peut ensuite générer un fichier **XML** selon une structure différente, spécialisée pour un environnement de développement donné, par exemple pour *Unity Pro* de Schneider :

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<FBExchangeFile xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:noNamespaceSchemaLocation="FBExchangeFile.xsd">
	<fileHeader company="Schneider Automation" product="Unity Pro L
V2.3-SP1A - 61127A" dateTime="date_and_time#2007-03-21-18:47:4"
content="Fichier source bloc fonctions" DTDVersion="6"></fileHeader>
	<contentHeader name="Station" version="0.0.000"></contentHeader>
	<FBSource nameOfFBType="SignalFilter" version="0.01"
dateTime="dt#2007-10-25-18:46:28">
		<inputParameters>
			<variables name="e_S" typeName="BOOL">
				<attribute name="PositionPin"
value="1"></attribute>
			</variables>
			<variables name="e_Delay" typeName="INT">
				<attribute name="PositionPin"
value="2"></attribute>
			</variables>
		</inputParameters>
		<outputParameters>
			<variables name="s_SU" typeName="BOOL">
				<attribute name="PositionPin"
value="1"></attribute>
			</variables>
			<variables name="s_SD" typeName="BOOL">
				<attribute name="PositionPin"
value="2"></attribute>
			</variables>
			<variables name="s_S" typeName="BOOL">
				<attribute name="PositionPin"
value="3"></attribute>
			</variables>
		</outputParameters>
		<publicLocalVariables>
			<variables name="_memSU"
typeName="BOOL"></variables>
			<variables name="_memSD"
typeName="BOOL"></variables>
```

```
                </publicLocalVariables>
                <privateLocalVariables>
                        <variables name="TIMERD"
typeName="TON"></variables>
                        <variables name="TIMERU"
typeName="TON"></variables>
                        <variables name="top" typeName="TopD"></variables>
                </privateLocalVariables>
                <FBProgram name="body">
                        <STSource>
                                top(e_S:=e_S);
                                if top.s_SU then
                                        _memSD:=0;
                                        _memSU:=1;
                                else if top.s_SD then
                                        _memSD:=1;
                                        _memSU:=0;
                                end_if;
                                TimerD (PT:=e_Delay, IN:=_memSD);
                                TimerU (PT:=e_Delay, IN:=_memSU);
                                s_SU:=TimerU.Q;
                                s_SD:=TimerD.Q;
                        </STSource>
                </FBProgram>
        </FBSource>
</FBExchangeFile>
```

CHAPITRE 8 :

CONCLUSIONS ET PERSPECTIVES

La difficulté majeure rencontrée par les entreprises depuis quelques années est due à la variabilité et à la richesse croissantes du contexte de production. D'une part, la clientèle est toujours plus exigeante dans ses cahiers des charges, les systèmes devant répondre à des fonctionnalités toujours plus nombreuses et complexes, d'autre part ces mêmes systèmes doivent être réalisés dans des laps de temps qui se réduisent, concurrence oblige.

Cette tendance qui s'est d'abord manifestée pour des biens de consommation concernant le grand public, s'est également étendue aux fournitures de matériel industriel. Dans notre contexte des systèmes automatisés, les concepteurs cherchent des moyens de répondre à cette situation. On met l'accent sur la réutilisabilité des développements dans des contextes différents. La logique de développement en spécifications successives d'un système, par les différents corps de métiers impliqués, montre ses limites car la priorité est souvent la rapidité de spécification du système global au détriment du temps consacré à une réflexion sur la composition d'éléments fonctionnels plus indépendants.

Cependant, la spécificité des technologies mises en oeuvre dans le cadre des systèmes automatisés, comme nous l'avons souligné dans les chapitres 2 et 3 de ce manuscrit, a contribué à ralentir l'évolution des méthodologies de conception dans ce domaine.

Le chapitre 4 souligne l'importance d'une démarche de modélisation lors des étapes de spécification, puis présente un état de l'art des méthodologies pouvant être utilisées dans le cadre des systèmes automatisés mais relève aussi les lacunes de ces approches traditionnelles vis-à-vis de la problématique initiale en terme de réactivité aux évolutions des contextes de conception.

L'approche objet constitue justement une réponse pertinente à ces contraintes comme l'explique le chapitre 5. La conception est alors centrée sur la spécification d'éléments de façon indépendante pour une composition ultérieure lors de la réalisation d'un système. Cette approche est notamment issue du domaine de l'informatique et a inspiré de nouveaux courants de spécification de programmes. L'**OMG** est un consortium initié par cet élan et qui a proposé de nombreux outils, **CORBA,** et des langages comme **UML** visant à favoriser l'interopérabilité des développements et des systèmes, mais également la standardisation des représentations de systèmes orientés objet.

L'originalité du travail a été de transposer ces approches issues du monde de l'informatique au domaine des systèmes automatisés et de proposer une démarche de conception dirigée par les modèles. Ces derniers se focalisent sur la représentation d'entités élémentaires comprenant les spécifications de différents contextes de réalisation. Les corps de métiers ne travaillent plus de façon linéaire sur un même système global, avec des allers retours entre les différents services selon les travaux ou les retouches à effectuer, mais se concentrent sur la description d'une entité élémentaire de composition sous l'angle technique qui est le leur.

Nous proposons une mise en oeuvre de la conception avec deux courants de spécifications parallèles mais de sens opposés. Dans le premier on se concentre sur la description des briques élémentaires indépendamment d'un contexte donné, dans le second on agence ces briques pour composer le système global.

La méthodologie proposée donne un guide d'utilisation de diagrammes **UML** pour la spécification de ces deux approches et nous abordons le déploiement en

proposant un moyen de générer la logique de contrôle des systèmes automatisés en passant par le standard de représentation **XML** des programmes **IEC 61131** décrit par l'organisation **PLCopen**.

8.1 La répartition des rôles

Actuellement on retrouve souvent le schéma suivant, quelques personnes qui se concentrent sur la maintenance et la mise à jour des standards internes à l'entreprise, un autre groupe de personnes qui conçoit et spécifie le système en utilisant les standards internes tels quels puis un dernier groupe qui réalise et teste le système.

On note une évolution des rôles qui, d'abord visible dans les grandes industries, se généralise maintenant dans les **PME/PMI** (*Petites et Moyennes Entreprises/Industries*), à savoir le déplacement des efforts de travail de la réalisation vers la spécification. On investit plus de temps et de moyens pour les phases de recherche et de conception au détriment des étapes de réalisation. L'externalisation des étapes de réalisation s'observe de plus en plus dans les entreprises qui se cantonnent à imaginer, spécifier, puis sous-traiter la réalisation voire l'assemblage avant de récupérer le système pour les essais de fonctionnement.

Pour les systèmes automatisés, les automaticiens auront de moins en moins à concevoir l'intégralité du code des programmes. Les éléments contrôlant des parties réalisées par la sous-traitance auront généralement une logique de commande déjà définie par le fournisseur, le seul vrai connaisseur des spécificités de l'élément intégré. Les automaticiens auront à insérer les différentes commandes des sous-parties d'ensembles et à les coordonner avec des blocs de programme métiers, réalisés en interne par les personnes chargées de la spécification du standard automatisme.

L'automaticien, chargé de l'implémentation d'un programme, devient donc un architecte dans le sens où il doit assembler des éléments de commande entre eux, de façon à obtenir le fonctionnement désiré. Quant aux automaticiens missionnés pour la réalisation des standards métiers, ils devront, comme précisé dans ce rapport, se focaliser sur des modules de commande élémentaires et les spécifier indépendamment d'un contexte de réalisation ou même d'un environnement de développement donné. La

standardisation des langages de spécification et l'utilisation de profils technologiques ,permettant de passer simplement du standard à un environnement donné, permettront de gagner du temps, notamment dans les métiers de l'intégration où le type et la marque des matériels évoluent selon les exigences client.

8.2 **L'évolution des matériels et des outils**

D'un côté la gamme des matériels informatiques s'étoffe considérablement au fil des années avec le perfectionnement des composants internes plus fiables et plus robustes (mécaniques non tournantes, dissipation de chaleur), permettant des alimentations de plus en plus élevées sans surchauffe, l'intégration des **PC** directement dans les armoires électriques, sans oublier l'augmentation de la taille des disques de type ***Compact Flash***, offrant la possibilité d'utiliser des applications plus riches (langages et bibliothèques de programmation plus nombreuses) et l'apport des technologies classiques au niveau informatique (par exemple un **OS** Windows avec l'ouverture associée pour la communication avec des niveaux informationels supérieurs). Le fossé entre les **PLC** et les ordinateurs industriels (***PC industriels***) se réduit donc en terme de robustesse même s'il reste une marge importante qui fait que les utilisateurs préfèrent encore conserver les premiers pour des applications à forte réactivité temporelle ou des fonctionnements dans des milieux très hostiles.

De l'autre côté, les automates évoluent également et s'enrichissent de nouvelles fonctionnalités, notamment dans leur ouverture vers les niveaux informationnels supérieurs de l'entreprise. L'intégration de *serveurs Web* embarqués permet la consultation et l'exploitation de données réelles par des clients extérieurs autorisés. Certains environnements de programmation proposent également des fonctions d'échange avec une base de données, ce qui permet de mettre directement à jour certaines données sans passer par une supervision. Au niveau des environnements de programmation, des progrès également importants pour la productivité du codage sont visibles, par exemple dans la gestion mémoire transparente (pas d'adressage physique à spécifier pour des données internes au programme) et l'intégration de concepts objet dans les langages structurés.

On voit donc que les deux domaines cherchent à combler leurs différentes

lacunes vis-à-vis des besoins des utilisateurs actuels, et on observe un rapprochement prudent mais inévitable des deux courants d'évolution. Il est probable que dans un futur proche le parc des plates-formes d'automatisme observera l'accélération de l'augmentation du nombre d'intégrations de matériels de type ***automate-PC*** ou ***PC industriels*** dans les architectures des systèmes automatisés. Ces éléments proposeront un ***OS temps réel*** permettant d'ordonnancer tâches critiques, pour la logique de commande, et tâches non-critiques pour la partie supervision ou traçabilité. Ils permettent un gain financier certain, en économisant l'achat d'une plate-forme supplémentaire pour la supervision sur un pupitre opérateur, et simplifient l'architecture automatisme.

Notre travail s'inscrit dans cette évolution, en proposant une spécification de la logique de commande et de contrôle au sein d'un même modèle de représentation, indépendamment d'une plate-forme donnée.

Au niveau des outils disponibles, l'évolution des ateliers logiciels tend à proposer des suites bureautiques composées de différents modules gérant les différents aspects d'une architecture de contrôle commande, communication, programme, interface opérateur. Une nouveauté récente du côté de Schneider par exemple est d'offrir la possibilité de définir des éléments de programmes liés à des objets graphiques, formant ainsi des *Modules Contrôle* (**ScoD**) possédant donc la spécification de la logique de commande et de la visualisation graphique d'un élément au niveau de la supervision. On a ainsi deux facettes réunies au sein d'un même *objet d'automatisme*. Notre approche de composant multi-facettes va plus loin mais correspond néanmoins à l'évolution actuelle des modes de conception dans les outils de programmation automate.

8.3 **Perspectives de travaux supplémentaires**

A l'avenir on peut donc prévoir un approfondissement de ce concept d'objet d'automatisme multi-facettes, et une évolution des modes de programmation en conséquence, avec la spécification d'un élément donné (par exemple un convoyeur) dans les différentes facettes identifiées, puis la connexion des différents éléments déterminés pour la réalisation d'un système plus global.

L'application de la logique **MDA** avec l'utilisation du langage de

modélisation UML, technologiquement neutre, permet de spécifier un premier niveau de définition dans les différentes facettes au sein d'un même modèle. L'extensibilité d'**UML** permet ensuite d'affiner les descriptions en fonction des contextes décrits. Nous avons présenté l'extension **SysML** qui nous semble particulièrement prometteuse pour décrire les logiques de fonctionnements particulières des automates, ainsi que les schémas de communication; la finalité étant la génération d'implémentations (code programme, description XML) lors du choix technologique final pour une facette donnée.

8.3.1 Décomposition des facettes

Nous avons décrit et choisi certaines facettes dans ce mémoire mais le travail d'identification est à poursuivre. Nous suggérons des réflexions supplémentaires pour affiner la spécification des facettes: Par exemple définir et isoler dans une sous-facette de la logique de fonctionnement, des caractéristiques propres à la sécurité, avec des mécanismes de contrôle-commande devant réagir dans le cadre de contraintes bien particulières, s'appuyer sur des travaux du domaine, comme pour modéliser des spécifications liées à la sécurité et à l'analyse du risque dans le domaine de la robotique [Guiochet 2003].

Le domaine de la maintenance et de la fiabilité représente également un aspect crucial lors de la définition de systèmes automatisés: isoler des spécifications particulières pour cette partie constitue un point très pertinent. De même pour les procédures de remise en marche après défaillance, il serait possible de réaliser un canevas générique de procédures et d'enchaînements d'actions à mener sur un système lors d'une erreur (visualisations, sons, validations d'interventions).

Des travaux sur l'extensibilité potentielle d'**UML** dans ces sous-facettes particulières pourrait représenter une piste non moins intéressante de recherche; les spécifications des extensions de ce langage sont en cours dans de nombreux domaines et tiennent compte des travaux et critiques des chercheurs extérieurs.

8.3.2 Efforts de Standardisation

Des travaux pourraient également être menés sur une représentation **XML** plus détaillée du langage **ST** de la norme **IEC 61131**, en continuité avec les spécifications proposées par le **TC6 PLCopen**. L'objectif est d'obtenir une correspondance plus précise entre les algorithmes représentés dans les diagrammes de comportement **UML** et la syntaxe utilisée dans le langage structuré automate.

CHAPITRE 9 :
CAS D'ÉTUDE, LE PRÉPARATEUR DE COUCHES ROBOTISÉ.

La machine qui a constitué le sujet d'étude concret pour la mise en place de ce travail de thèse est une partie d'un palettiseur robotisé. Elle est issue d'un brevet déposé et constitue une offre originale et performante sur le marché actuel des palettiseurs. Pour des raisons de confidentialité, la description de ce système est simplifiée dans ce rapport.

9.1 **Principe de fonctionnement**

DÉFINITION 7 : PALETTISEUR

Système mécanique automatisé réalisant l'organisation spatiale de produits sous forme d'un empilage structuré, stable et optimisé appelé Palette, en vue de leur transport ultérieur.

On retrouve différentes phases dynamiques décomposant la fonction de

palettisation:

1. La préparation du produit en amont de la machine, phase qui consiste à adapter les caractéristiques physiques et dynamiques du produit pour qu'il puisse poursuivre correctement son parcours dans les autres phases de traitement. On retrouve ici par exemple la modification de sa vitesse de déplacement à l'aide du convoyage, la rotation du produit, l'isolation du produit par rapport aux autres.

2. Le passage du produit ainsi préparé vers une zone de réalisation de couche.

3. Le positionnement du produit par rapport à d'autres produits qui vont constituer une couche de produits. Cette phase correspond à la réalisation d'une couche.

4. L'évacuation de la couche ainsi réalisée.

5. Le positionnement de la couche par rapport aux autres couches, le tout constituant un empilement structuré de produits. On retrouve également dans cette phase l'insertion ou non au niveau de l'empilement d'intercalaires ou de supports en bois.

6. L'évacuation de la palette ainsi constituée vers un module de traitement amont.

Le préparateur de couches robotisé réalise les fonctions 1 à 4 de l'énumération précédente et est représenté schématiquement sur la Figure 9.1.

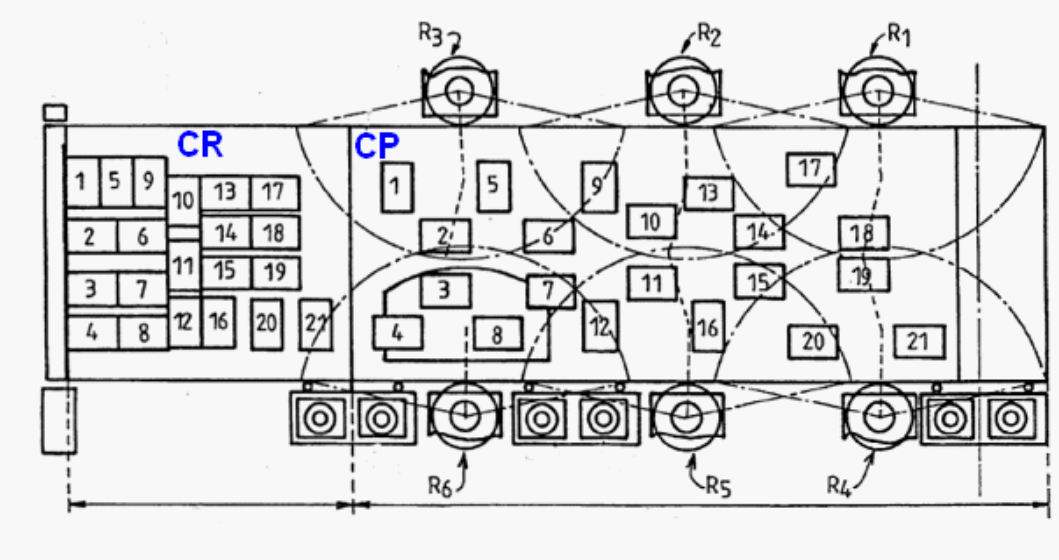


Figure 9.1.: Plan du préparateur de couche robotisé.

Les rectangles numérotés représentent des produits, le sens de déplacement est de la droite vers la gauche. Les cercles annotés R1 à R6 correspondent aux emplacements des différents robots intervenant dans la réalisation du procédé. Les demi-cercles représentent leurs zones d'actions respectives. Certains possèdent des supports d'outils représentés par des rectangles avec deux cercles à l'intérieur, et permettent aux robots le changement de l'outil selon le type de produit à traiter. Deux zones sont représentées avec des colis en leur sein:

- la zone **CP** (Convoyeur de Préparation) sur laquelle s'effectuent les déplacements et les orientations de produits par les robots.
- la zone **CR** (Convoyeur de Récupération) possédant à son extrémité gauche une butée d'accumulation stoppant la progression et le déplacement des produits de la droite vers la gauche.

Les produits numérotés de 1 à 21 correspondent à une couche, c'est à dire à un niveau dans l'empilement final de la palette. On voit que cette couche est constituée par accumulation des produits précédemment positionnés de façon optimale pour donner l'organisation de produits désirée pour la couche en cours. Les derniers trous présents entre les produits seront comblés dans une phase ultérieure appelée phase de conformation.

9.1.1 Problématique de flexibilité

L'une des caractéristiques de la machine est qu'elle peut s'adapter à une grande plage de cadences et de contextes d'utilisation. La partie générant les données de contrôle doit pouvoir se conformer à ces différentes configurations et permettre un paramétrage rapide de l'installation lorsqu'elle évolue. On peut par exemple avoir de une à *n* voies d'alimentation, de un à *n* robots, des produits de n'importe quelle forme qu'il est parfois possible de regrouper entre eux afin d'améliorer le rythme de production lors de la manipulation par les robots.

L'approche par composants multi-facettes répond donc parfaitement à cette contrainte de flexibilité dans l'évolution du système étudié. Les différents éléments constituant le préparateur de couches robotisé ont été définis comme des composants individuels possédant différentes caractéristiques mécaniques, de contrôle-commande, de communication. L'ajout ou la suppression d'un élément entraîne la mise à jour des conteneurs (notamment pour l'interface de dialogue et la logique de commande) et des données de fonctionnement de façon automatique grâce à une conception orientée composants.

9.1.2 Évolution des approches de spécifications pour l'automatisme

Le travail de thèse a permis de lancer une nouvelle dynamique pour la spécification des systèmes conçus au sein de l'entreprise, en particulier au niveau des programmes automatisme. Des efforts sont en effet mis en oeuvre au niveau de définition d'éléments de contrôle pouvant être utilisés dans différents contextes de fonctionnement, en suivant les étapes décrites dans ce mémoire. Une bibliothèque est ainsi constituée petit à petit pour permettre, à terme, une élaboration plus structurée des logiques de contrôle complexes qui pourront de plus être amenées à évoluer selon les demandes client (nous ne présentons pas les blocs fonction issus de ces réflexions pour des raisons de confidentialité).

Lors de réunions régulières, un groupe de travail se focalisant sur l'**AOM** décrite précédemment, réfléchit sur un ensemble d'unités de contrôle en se détachant au

maximum d'un contexte de projet en cours dans l'entreprise. Durant une autre étape, l'intégration des éléments ainsi standardisés est réalisée de façon à éclaircir et à restructurer les programmes déjà existants. L'objectif est d'améliorer l'existant sans avoir à tout reconstruire, ce qui représenterait un coût trop important pour une moyenne entreprise comme Newtec.

Par contre pour les nouvelles machines à venir, l'objectif est d'intégrer complètement la démarche de définition par objets d'automatisme, avec une logique de fonctionnement basée sur la connexion de blocs fonction élémentaires.

9.1.3 Le découpage logiciel en composants

En parallèle aux efforts de standardisation par blocs de la partie automatisme, un travail est mené sur les éléments de contrôle logiciel permettant le paramétrage des éléments métier et des éléments physiques, la simulation du fonctionnement et le stockage des informations liées à une campagne de production.

L'approche par composants couplée avec la méthodologie proposée dans nos travaux a conduit à la réalisation d'un grand nombre de composants logiciels, permettant la définition des paramètres métiers (tels que les programmes de palettisation) indépendamment d'une machine donnée.

Le calcul des paramètres dynamiques se fait en fonction des éléments physiques ajoutés dans l'implantation globale (avec l'ajout des composants logiciels associés) et les écrans de simulation ou de supervision, conçus comme des conteneurs graphiques génériques, permettent l'intégration automatique des composants ainsi chargés (technologie COM/DCOM utilisée, .NET).

9.1.4 Validation par simulation

Nous avons spécifié les facettes de logique de fonctionnement par objet d'automatisme sans préciser :

- d'où proviennent les entrées
- à quoi sont reliées les sorties

– quelle est la syntaxe du code d'implémentation liée à la plate-forme d'exécution

Il est donc possible de simuler sur une plate-forme de notre choix chaque élément en lui fournissant les entrées nécessaires (par exemple, en utilisant le simulateur d'un environnement de développement d'automatisme pour tester des blocs fonction). Ainsi, dans notre cas d'étude, la simulation permet de reproduire le comportement de la logique de fonctionnement finale à partir d'évènements artificiellement générés, ce qui permet d'avoir une vision du fonctionnement proche du comportement réel au niveau des réactions des éléments physiques.

La dynamique des produits lors de la simulation n'est pas gérée par les composants physiques. Chaque produit possède son propre modèle de comportement dynamique, comprenant un ensemble de paramètres de stabilité selon sa forme, son poids, sa rigidité et les frottements qu'il occasionne en déplacement.

Ainsi, des sources de produits, avec des lois de création statistiques données éventuellement par un client, sont-elles créées dans l'environnement de simulation; ensuite les colis évoluent, dans l'environnement, en fonction de leur caractéristiques cinématiques et des réactions des différents éléments physiques rencontrés.

Cette gestion a l'avantage de réduire la complexité de la gestion globale en incorporant, au niveau des objets, des comportements dynamiques propres. De plus, on obtient ainsi une simulation très proche de ce qui se passera dans la réalité, en évitant des erreurs de positionnement et surtout des interférences physiques non désirées et potentiellement coûteuses.

Cette démarche, s'appuyant beaucoup sur le renseignement de paramètres dynamiques *produit* pas forcément connus à l'avance, fait que le modèle des produits est amené à évoluer et à se préciser au fur et à mesure des essais réels.

9.1.5 Vue Globale

Au niveau des spécifications plus générales, on regroupe donc dans la définition d'un module physique, les éléments de définition mécanique (nomenclature,

définitions géométriques, contraintes physiques), les éléments de définition automatisme (blocs, diagrammes blocs fonction) et les éléments logiciels (IHM à intégrer dans les conteneurs, algorithmes de paramétrage).

Nous nous efforçons dans une activité nouvelle pour l'entreprise, de réunir ces spécifications de façon cohérente au sein d'un même modèle graphique en UML, conformément à l'approche développée dans le travail de cette thèse.

9.2 **Exemple de réalisation suivant la méthodologie de conception décrite dans ce mémoire**

L'ajout des paragraphes ci-dessous fait suite à une requête des rapporteurs de ce mémoire. Elle présente des détails supplémentaires sur des éléments de réalisation. Les informations contenues dans cette partie sont strictement confidentielles et destinées uniquement à la lecture par les rapporteurs et tuteurs, elles ne seront pas diffusées dans la version finale du manuscrit de thèse.

Les diagrammes et éléments de code décrivent un sous-ensemble du « *préparateur de couches robotisé* » présenté dans le mémoire: la gestion du cadencement des produits en amont de la table de préparation.

9.2.1 Le cadencement

Le schéma ci-dessous reprend la description du cadencement :

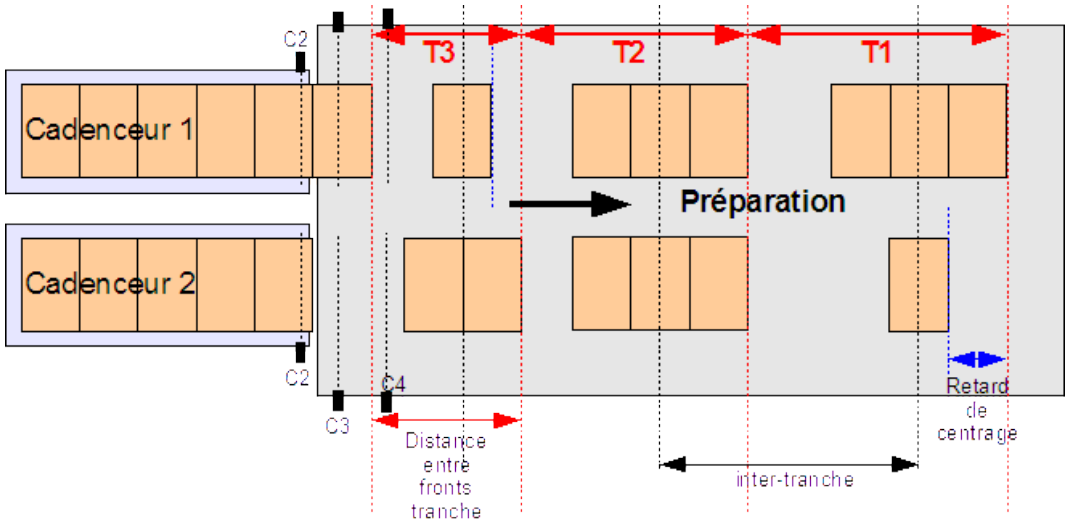


Figure 9.2: Schéma

Le cadencement correspond à l'envoi successif de produits sur le convoyeur de préparation avec des espacements précis et de façon à avoir les groupes de colis centrés dans les « *tranches* ».

Ce fonctionnement met en œuvre :

- Des convoyeurs commandés en position (moteurs Brushless): les cadenceurs.
- Une cellule d'arrêt C2 pour chaque cadenceur permettant de stopper le cadenceur lors de son remplissage.
- Une cellule de synchronisation C3 effectuant le contrôle de longueur des groupes de produits et le contrôle de synchronisme dans l'envoi des groupes d'une même tranche de produits.
- Une cellule C4 qui envoie des tops aux robots devant traiter les produits détectés

Dans un premier temps nous décrivons des briques élémentaires issues de l'approche montante d'analyse. Ces briques élémentaires sont réalisés en-dehors d'un

contexte de machine. Dans un second temps, nous présentons la brique composite de contrôle de cadencement, identifiée lors de l'analyse descendante pour la réalisation de la machine, et s'appuyant sur des briques élémentaires déjà existantes ou devant être créées.

9.2.1.1 Structure du modèle UML

Nous présentons ici les briques élémentaires « *convoyeur* » (*nConvoyeur*), « *convoyeur Brushless* » (*nConvBrush3dp* dérivé de *nConvoyeur*), « *capteur de proximité* » (*nCapteur*) et « *robot 4 axes* » (*nRobot4A*) correspondant à des objets d'automatisme élémentaires.

Le modèle a été réalisé dans différents outils de modélisation: *Poséidon*, *StarUML* puis *Visual Paradigm*. Les visualisations qui suivent ont été tirées de *StarUML*. Pour pouvoir générer des fichiers à partir du modèle, les plugins pour **XMI** et **C++** ont été inclus, ainsi que les profils **C++**. Les vues **SysML** ont été réalisés à l'aide d'extensions personnalisées dans l'entreprise via les stéréotypes mais ne sont pas réalisables tels quels pour le moment, en attendant que l'extension soit plus répandue.

Chaque brique est représentée par un élément « *package* » et contient des sous-packages correspondant aux différentes facettes de l'objet d'automatisme tel que nous l'avons défini précédemment (notamment Logique, Physique, UI (interface utilisateur), Communication, MES).

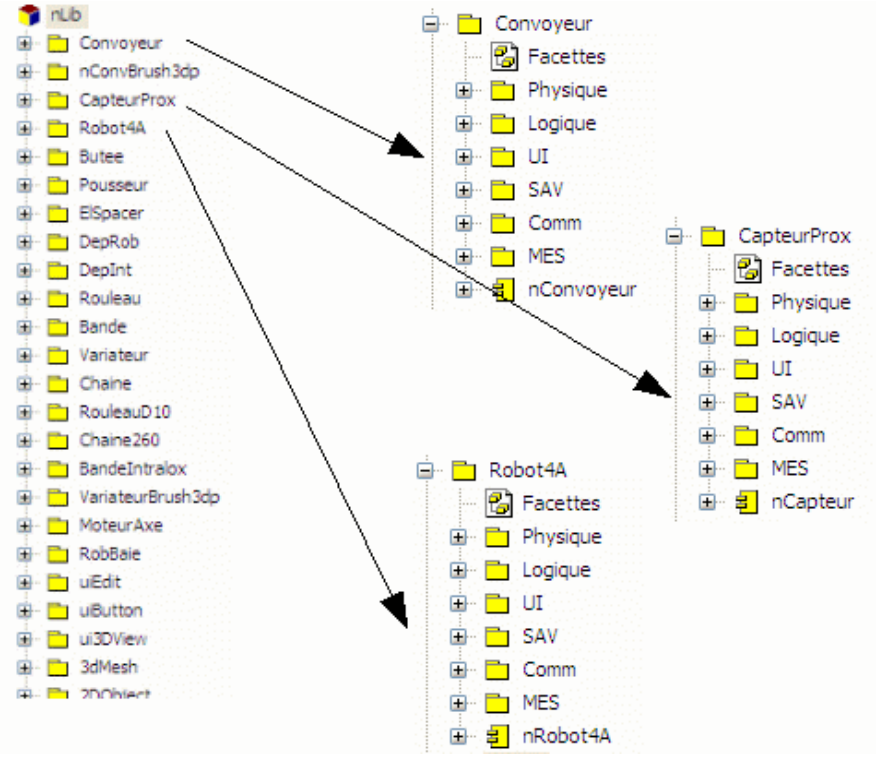


Figure 9.3: Structure du modèle UML

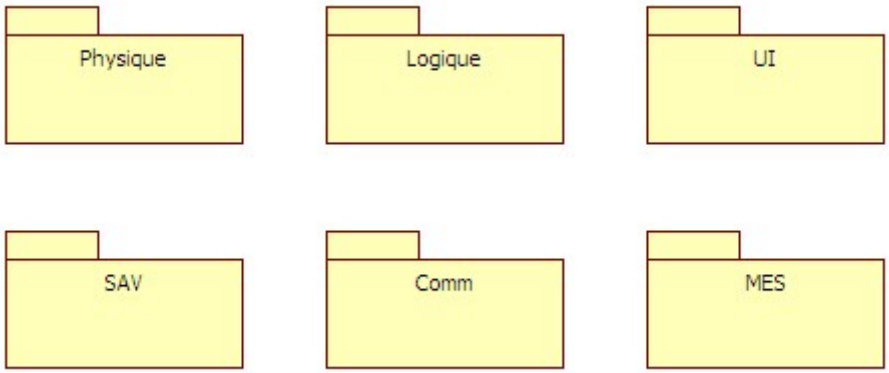


Figure 9.4: Vue des packages liés aux facettes

Chaque package contient l'ensemble des modèles et des éléments décrivant la facette concernée.

9.2.2 Brique élémentaire « *nConvoyeur* »

9.2.2.1 Facette Physique

Cette facette décrit deux aspects de la description physique du composant:

- Sa composition (sorte de nomenclature)
- Une vue physique interne permettant de visualiser les relations physiques entre les éléments internes et la correspondance de ces éléments avec l'extérieur de la brique décrite.

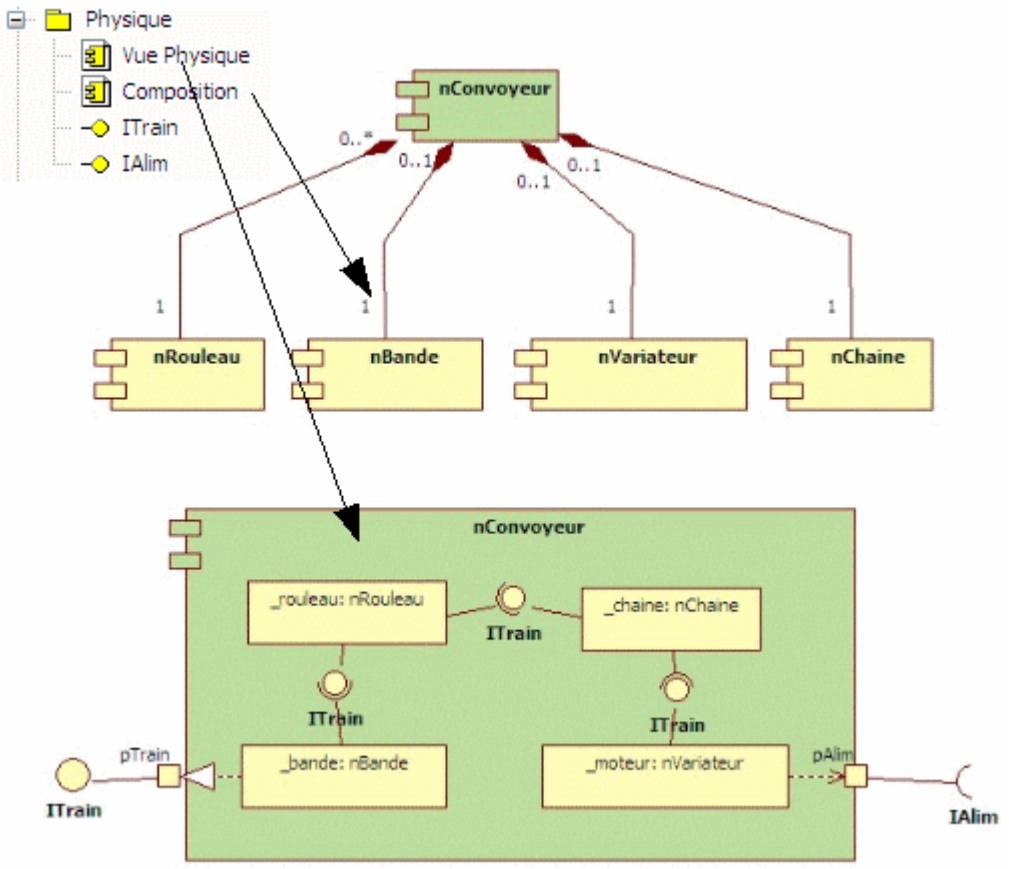


Figure 9.5: Modélisation de la facette physique

Sur le diagramme de composition, on peut voir qu'un convoyeur est composé d'un ensemble de rouleaux, d'une bande, d'un variateur et d'une chaîne. Ces éléments sont

également des briques élémentaires.

Sur la vue composite, les relations entre ces éléments sont illustrées via des interfaces. *ITrain* est une interface représentant l'entrainement mécanique entre les différents éléments. A partir d'une alimentation fournie par un objet extérieur via l'interface *IAlim,* le moteur fournit une force d'entrainement à la chaîne qui la communique à la bande via les rouleaux.

La figure ci-dessous montre comment le convoyeur peut-être spécialisé en fonction d'une utilisation précise :

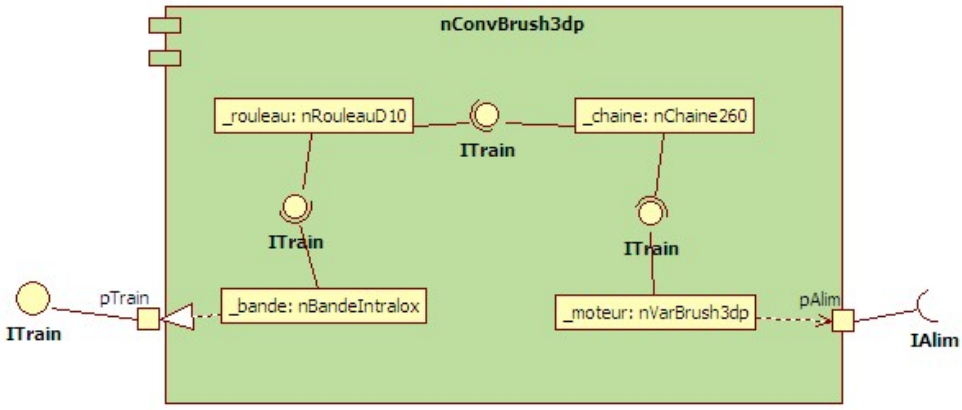


Figure 9.6: Spécialisation de nConvoyeur

9.2.2.2 Facette UI (interface utilisateur)

Cette facette précise notamment les éléments graphiques qui devront être présents sur l'interface de dialogue avec l'utilisateur. On y retrouve par exemple les champs d'édition définit par l'objet élémentaire *nUIEdit*. L'objet *nConvoyeur* pourra également être visualisé sous forme 2D (fenêtre 2D dérivée de 2DObject) ou 3D (objet dérivé de 3dMesh). Les visualisations 3D sont réalisées en DirectX dans nos applications et chaque brique que l'on voudra incorporer à la 3D devra spécifier les informations contenues dans 3dMesh.

De la même façon que pour la facette physique, on retrouve deux

diagrammes, l'un décrivant la composition de l'objet sous un angle interface utilisateur, et l'autre décrivant explicitement comment l'objet propose à l'extérieur des service d'interaction et de visualisation :

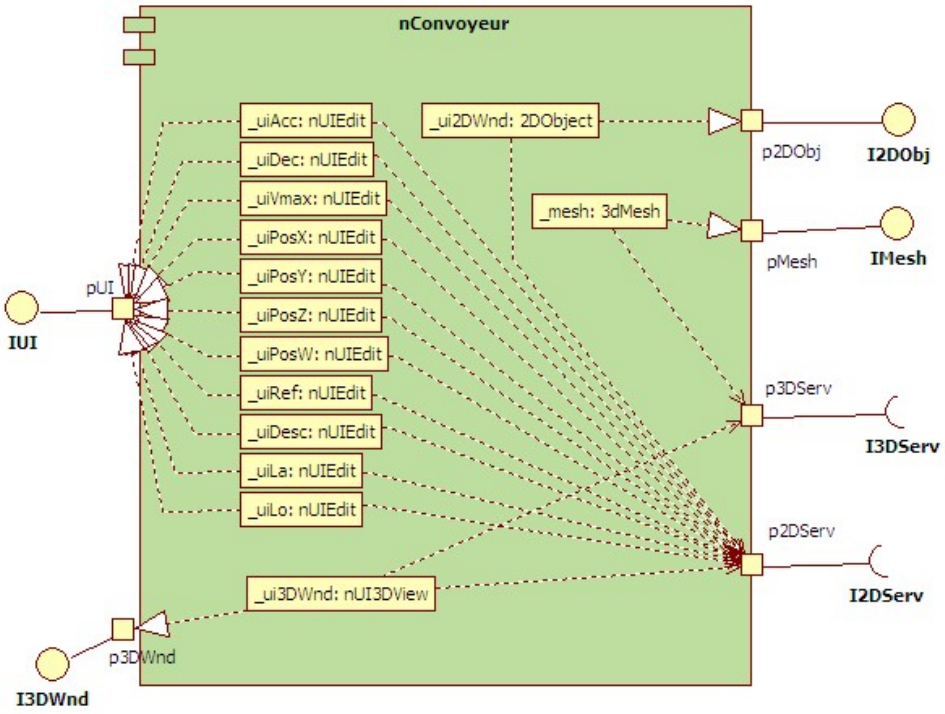


Figure 9.7: Vue Interface Utilisateur

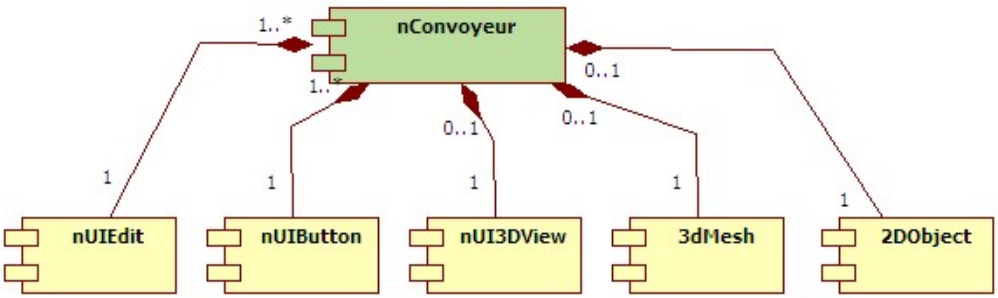


Figure 9.8: Composition de la facette UI

L'interface *IUI* propose une boîte de dialogue avec les champs de saisie, l'interface *I3DWnd* propose l'utilisation d'une fenêtre avec la visualisation 3D de l'objet *nConvoyeur*. L'interface *I2DObj* donne accès à l'objet 2D de *nConvoyeur* (une image par exemple). L'interface *IMesh* permet de récupérer l'objet 3D simple pour l'incorporer éventuellement dans une scène 3D plus complexe. Les interfaces *I3DServ* et *I2DServ* spécifient le besoin de services 2D et 3D (issus par exemple de bibliothèques ou de moteurs graphiques) pour que la facette UI de l'objet puisse fonctionner correctement.

9.2.2.3 Facette Logique

Nous exprimons ici le comportement logique de l'objet pour ce qui est du traitement des données. La première étape est une représentation en UML de l'objet sous forme de composant dans un diagramme de représentation de type « *boîte noire* ».

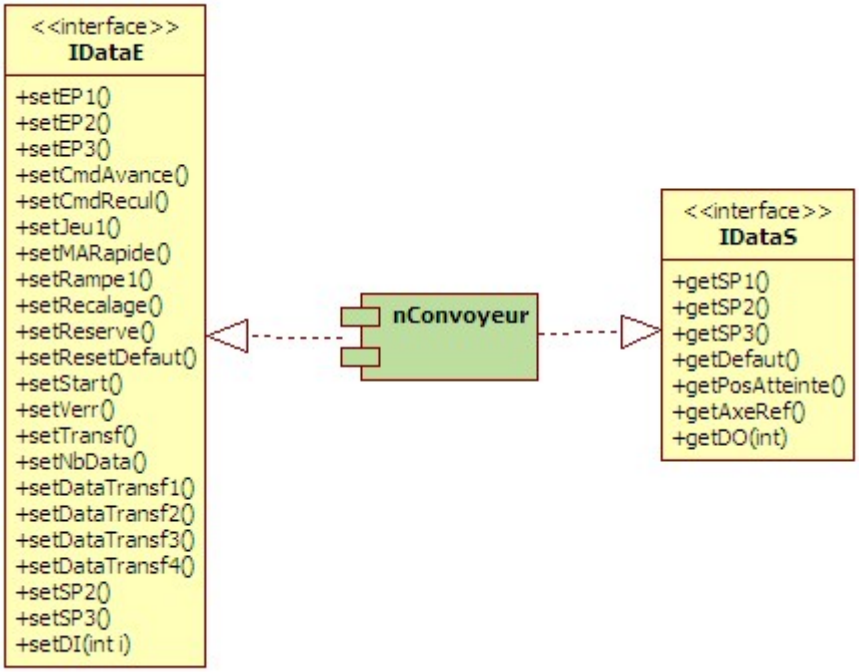


Figure 9.9: Diagramme de composant pour la vue logique

On voit deux interfaces, une en entrée et l'autre en sortie, permettant de décrire les données qu'il est possible de mettre à jour ou de récupérer, mais également de

décrire les évènements déclenchant l'activité de l'objet (*Start*). Nous mélangeons ici flux de données et flux de contrôle en les représentant à l'aide de services au sein d'une interface.

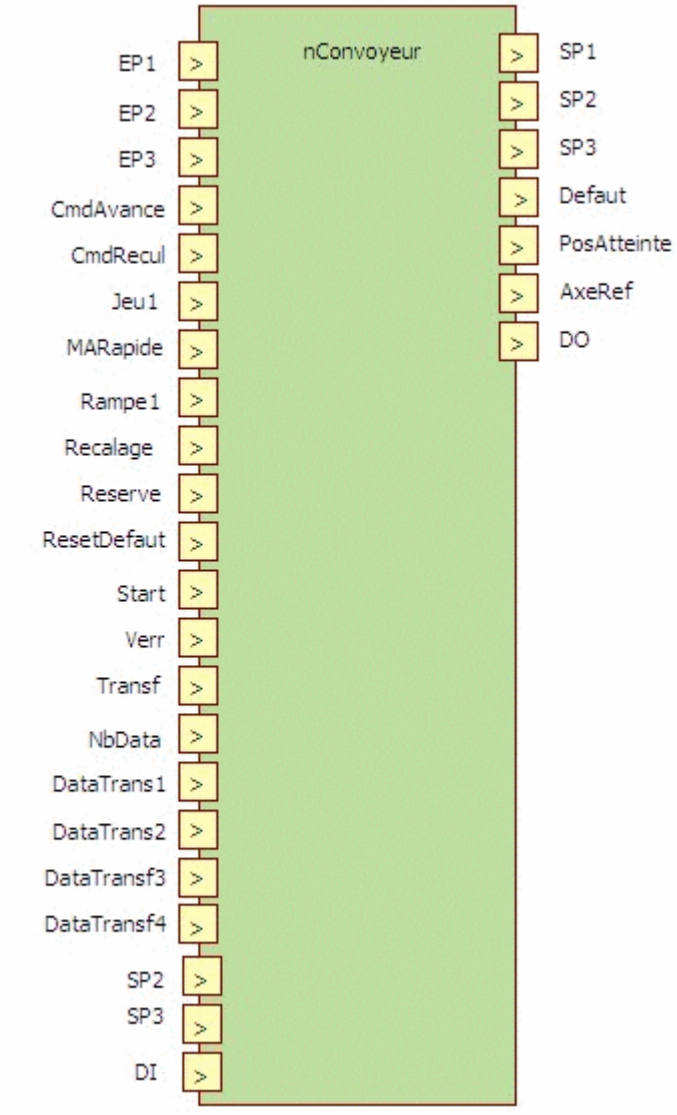


Figure 9.10: Diagramme de définition de bloc SysML de nConvoyeur

A partir de la figure 9.9, on étend le diagramme à l'aide des éléments proposés dans le langage **SysML**, notamment le diagramme de définition de

« *block* »(bdd). L'extension se fait manuellement mais selon la logique **MDA** il devrait être possible de « transformer » le modèle **UML** précédent en modèle **SysML** avec les mécanismes de transformations conformes à **MDA** (conversion en cours de développement). Pour le moment nous reprenons donc manuellement la description SysML pour obtenir le bloc de la figure 9.10.

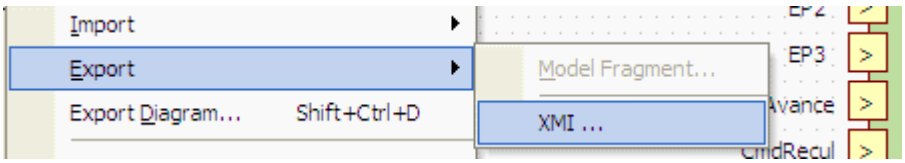
En entrée on retrouve les 3 entrées physiques *EP1*, *EP2*, *EP3* qui sont des mots récupérés directement à partir du variateur. Les entrées *Jeu1*, *MARapide*, *Rampe1, Recalage, Reserve, ResetDefaut, Verr* sont des bits contenus dans *EP1* et que l'on associe aux mêmes informations sur tous les convoyeurs dans le standard automatisme de l'entreprise. *Start* permet de lancer le démarrage du convoyeur. *Transf* permet de mettre le convoyeur en mode de transfert, pour écrire certaines informations dans le variateur du convoyeur avant le lancement du fonctionnement automatique. Le nombre de ces données de transfert est donné par *NbData*, et les données sont spécifiées dans *DataTransf* 1 à 4. *SP2* et *SP3* sont des consignes pour les sorties physiques *SP2* et *SP3*. *DI* correspond aux *Data Input*.

En sortie, on retrouve les sorties physiques 1 à 3 (pour écriture vers l'extérieur), un bit de défaut, de position atteinte, d'axe référencé ainsi que les *DO* (*Data Output*)

9.2.2.4 La génération de code

Nous nous attardons sur la vue Logique en précisant le mécanisme de génération de code utilisé pour obtenir un squelette de programme automate sous *Unity* (*TeleMecanique*).

A partir du diagramme de définition de bloc SysML précédent, nous utilisons l'exportation **XMI** sur le « *block* » *nConvoyeur* pour obtenir un premier fichier de description sous forme **XML**.



Le fichier est sous cette forme et décrit le composant, avec les éléments UML (modèle, package, commentaire, namespace) et SysML (block, FlowPort (étendu du port UML 2.0 pour les flux de données) atomiques (une seule variable par port) « *in* » ou « *out* ») :

```
<?xml version = "1.0" encoding = "UTF-8"?>
<XMI xmi.version = "2.0" xmlns:UML="href://org.omg/UML/2.0"
xmlns:SysML="http://schema.omg.org/spec/SysML/1.0">
<XMI.header>
 <XMI.documentation>
  <XMI.owner></XMI.owner>
  <XMI.contact></XMI.contact>
  <XMI.exporter>SysMLXMI-Addin</XMI.exporter>
  <XMI.exporterVersion>1.0</XMI.exporterVersion>
  <XMI.notice></XMI.notice>
 </XMI.documentation>
 <XMI.metamodel xmi.name = "SysML" xmi.version = "1.0"/>
</XMI.header>
<XMI.content>
<UML:Model xmi.id="UMLProject.1">
 <UML:Namespace.ownedElement>
  <UML:Package xmi.id="UMLPackage.2" name="nConvoyeur" visibility="public"
isSpecification="false" namespace="UMLProject.1" isRoot="false" isLeaf="false"
isAbstract="false">
   <UML:Namespace.ownedElement>
                [.....................]//packages des autres facettes
    <UML:Package xmi.id="UMLPackage.4" name="Logique" visibility="public"
isSpecification="false" namespace="UMLPackage.2" isRoot="false" isLeaf="false"
isAbstract="false">
     <UML:Namespace.ownedElement>
      <SysML:Block xmi.id="SysMLBlock.5" name="nConvoyeur" visibility="public"
isSpecification="false" namespace="UMLPackage.4" clientDependency="UMLRealization.8
UMLRealization.12 UMLRealization.15" isRoot="false" isLeaf="false" isAbstract="false">
       <SysML:FlowPort direction="in" isAtomic="true" datatype="integer" name="eiEP1">
        <UML:Comment>Retour Etat</UML:Comment>
       </SysML:FlowPort>
       <SysML:FlowPort direction="in" isAtomic="true" datatype="integer" name="eiEP2">
        <UML:Comment>Retour Donnée Physique 2</UML:Comment>
       </SysML:FlowPort>
       <SysML:FlowPort direction="in" isAtomic="true" datatype="integer" name="eiEP3">
        <UML:Comment>Retour Donnée Physique 3</UML:Comment>
       </SysML:FlowPort>
       <SysML:FlowPort direction="in" isAtomic="true" datatype="boolean"
name="ebCmdAvance">
        <UML:Comment>bit Avance</UML:Comment>
       </SysML:FlowPort>
       <SysML:FlowPort direction="in" isAtomic="true" datatype="boolean"
name="ebCmdRecul">
        <UML:Comment>bit Recul</UML:Comment>
```

```
        </SysML:FlowPort>
        <SysML:FlowPort direction="in" isAtomic="true" datatype="boolean"
name="ebJeu1">
          <UML:Comment>bit Jeu</UML:Comment>
        </SysML:FlowPort>
        <SysML:FlowPort direction="in" isAtomic="true" datatype="boolean"
name="ebMARapide">
          <UML:Comment>bit MA Rapide</UML:Comment>
        </SysML:FlowPort>
        <SysML:FlowPort direction="in" isAtomic="true" datatype="boolean"
name="ebRampe1">
          <UML:Comment>bit Rampe 1</UML:Comment>
        </SysML:FlowPort>
        <SysML:FlowPort direction="in" isAtomic="true" datatype="boolean"
name="ebRecalage">
          <UML:Comment>bit Recalage</UML:Comment>
        </SysML:FlowPort>
        <SysML:FlowPort direction="in" isAtomic="true" datatype="boolean"
name="ebReserve">
          <UML:Comment>bit Reserve</UML:Comment>
        </SysML:FlowPort>
        <SysML:FlowPort direction="in" isAtomic="true" datatype="boolean"
name="ebResetDefaut">
          <UML:Comment>bit Reset Defaut</UML:Comment>
        </SysML:FlowPort>
        <SysML:FlowPort direction="in" isAtomic="true" datatype="boolean"
name="ebStart">
          <UML:Comment>bit Start</UML:Comment>
        </SysML:FlowPort>
        <SysML:FlowPort direction="in" isAtomic="true" datatype="boolean"
name="ebVerr">
          <UML:Comment>bit Verr</UML:Comment>
        </SysML:FlowPort>
        <SysML:FlowPort direction="in" isAtomic="true" datatype="boolean"
name="ebTransf">
          <UML:Comment>bit Transfert</UML:Comment>
        </SysML:FlowPort>
        <SysML:FlowPort direction="in" isAtomic="true" datatype="integer"
name="eiNbData">
          <UML:Comment>bit Nombre Données Physiques pour Transfert</UML:Comment>
        </SysML:FlowPort>
        <SysML:FlowPort direction="in" isAtomic="true" datatype="integer"
name="eiDataTransf1">
          <UML:Comment>Données de Transfert 1</UML:Comment>
        </SysML:FlowPort>
        <SysML:FlowPort direction="in" isAtomic="true" datatype="integer"
name="eiDataTransf2">
          <UML:Comment>Données de Transfert 2</UML:Comment>
        </SysML:FlowPort>
        <SysML:FlowPort direction="in" isAtomic="true" datatype="integer"
name="eiDataTransf3">
          <UML:Comment>Données de Transfert 3</UML:Comment>
        </SysML:FlowPort>
```

```
        <SysML:FlowPort direction="in" isAtomic="true" datatype="integer"
name="eiDataTransf4">
          <UML:Comment>Données de Transfert 4</UML:Comment>
        </SysML:FlowPort>
        <SysML:FlowPort direction="in" isAtomic="true" datatype="integer" name="eiSP2">
          <UML:Comment>Donnée Physique 2</UML:Comment>
        </SysML:FlowPort>
        <SysML:FlowPort direction="in" isAtomic="true" datatype="integer" name="eiSP3">
          <UML:Comment>Donnée Physique 3</UML:Comment>
        </SysML:FlowPort>
        <SysML:FlowPort direction="in" isAtomic="true" datatype="boolean"
name="ebDI15">
          <UML:Comment>bit Etat 15</UML:Comment>
        </SysML:FlowPort>
        <SysML:FlowPort direction="in" isAtomic="true" datatype="boolean"
name="ebDI16">
          <UML:Comment>bit Etat 16</UML:Comment>
        </SysML:FlowPort>
        <SysML:FlowPort direction="in" isAtomic="true" datatype="boolean"
name="ebDI17">
          <UML:Comment>bit Etat 17</UML:Comment>
        </SysML:FlowPort>
        <SysML:FlowPort direction="out" isAtomic="true" datatype="integer" name="siSP1">
          <UML:Comment>Etat</UML:Comment>
        </SysML:FlowPort>
        <SysML:FlowPort direction="out" isAtomic="true" datatype="integer" name="siSP2">
          <UML:Comment>Donnée Physique 2</UML:Comment>
        </SysML:FlowPort>
        <SysML:FlowPort direction="out" isAtomic="true" datatype="integer" name="siSP3">
          <UML:Comment>Donnée Physique 3</UML:Comment>
        </SysML:FlowPort>
        <SysML:FlowPort direction="out" isAtomic="true" datatype="boolean"
name="sbDefaut">
          <UML:Comment>bit Defaut</UML:Comment>
        </SysML:FlowPort>
        <SysML:FlowPort direction="out" isAtomic="true" datatype="boolean"
name="sbPosAtteinte">
          <UML:Comment>bit Destination Atteinte</UML:Comment>
        </SysML:FlowPort>
        <SysML:FlowPort direction="out" isAtomic="true" datatype="boolean"
name="sbAxeRef">
          <UML:Comment>bit Axe reference</UML:Comment>
        </SysML:FlowPort>
        <SysML:FlowPort direction="out" isAtomic="true" datatype="boolean"
name="sbDO1">
          <UML:Comment>bit 1 Etat</UML:Comment>
        </SysML:FlowPort>
        <SysML:FlowPort direction="out" isAtomic="true" datatype="boolean"
name="sbDO2">
          <UML:Comment>bit 2 Etat</UML:Comment>
        </SysML:FlowPort>
        <SysML:FlowPort direction="out" isAtomic="true" datatype="boolean"
```

```
name="sbDO3">
            <UML:Comment>bit 3 Etat</UML:Comment>
          </SysML:FlowPort>
          <SysML:FlowPort direction="out" isAtomic="true" datatype="boolean"
name="sbDO4">
            <UML:Comment>bit 4 Etat</UML:Comment>
          </SysML:FlowPort>
          <SysML:FlowPort direction="out" isAtomic="true" datatype="boolean"
name="sbDO6">
            <UML:Comment>bit 6 Etat</UML:Comment>
          </SysML:FlowPort>
          <SysML:FlowPort direction="out" isAtomic="true" datatype="boolean"
name="sbDO7">
            <UML:Comment>bit 7 Etat</UML:Comment>
          </SysML:FlowPort>
          <SysML:FlowPort direction="out" isAtomic="true" datatype="boolean"
name="sbDO10">
            <UML:Comment>bit 10 Etat</UML:Comment>
          </SysML:FlowPort>
          <SysML:FlowPort direction="out" isAtomic="true" datatype="boolean"
name="sbDO11">
            <UML:Comment>bit 11 Etat</UML:Comment>
          </SysML:FlowPort>
          <SysML:FlowPort direction="out" isAtomic="true" datatype="boolean"
name="sbDO12">
            <UML:Comment>bit 12 Etat</UML:Comment>
          </SysML:FlowPort>
          <SysML:FlowPort direction="out" isAtomic="true" datatype="boolean"
name="sbDO13">
            <UML:Comment>bit 13 Etat</UML:Comment>
          </SysML:FlowPort>
          <SysML:FlowPort direction="out" isAtomic="true" datatype="boolean"
name="sbDO14">
            <UML:Comment>bit 14 Etat</UML:Comment>
          </SysML:FlowPort>
          <SysML:FlowPort direction="out" isAtomic="true" datatype="boolean"
name="sbDO15">
            <UML:Comment>bit 15 Etat</UML:Comment>
          </SysML:FlowPort>
          <SysML:FlowPort direction="out" isAtomic="true" datatype="boolean"
name="sbDO16">
            <UML:Comment>bit 16 Etat</UML:Comment>
          </SysML:FlowPort>
        </SysML:Block>

        <UML:Interface>
                [.....]//description des interfaces sur la vue boîte noire du composant
        </UML:Interface>
      </UML:Namespace.ownedElement>
    </UML:Package>
  </UML:Namespace.ownedElement>
 </UML:Package>
</UML:Namespace.ownedElement>
```

```
</UML:Model>
</XMI.content>
</XMI>
```

Dans l'étape suivante nous cherchons à effectuer une première transformation du fichier pour le mettre sous un format conforme à la structure *XML* **PLCopen** de description des structures automates, en l'occurrence une unité d'organisation de programme (*POU*) de type bloc fonctionnel (*functionBlock*). Nous avons également scindé le bloc en deux blocs pour séparer la lecture ou la mise à jour des données physiques des données utilisées par le programme. Enfin, à ce niveau nous ajoutons le programme en langage structuré, nous verrons plus loin qu'il est issu d'une spécification au sein d'un diagramme d'activité **SysML**.

Bloc préliminaire :

```
<pou name="nConvoyeurE" pouType="functionBlock">
 <interface>
        <documentation>bloc de commande entrée Brushless 3dp</documentation>
        <inputVars>
                <variable name="eiEP1">
                        <type><INT/></type>
                        <documentation>Retour Etat</documentation>
                </variable>
                < variable name="eiEP2">
                        <type><INT/></type>
                        <documentation>Retour Donnée Physique 2</documentation>
                </variable>
                < variable name="eiEP3">
                        <type><INT/></type>
                        <documentation>Retour Donnée Physique 3</documentation>
                </variable>
        </inputVars>
        <outputVars>
                <variable name="siEP1" >
                        <type><INT/></type>
                        <documentation>Etat</documentation>
                </variable>
                <variable name="siEP2" >
                        <type><INT/></type>
                        <documentation>Données Physique 2</documentation>
                </variable>
                <variable name="siEP3">
                        <type><INT/></type>
                        <documentation>Données Physique 2</documentation>
                </variable>
                <variable name="sbDO1">
```

```
        <type><BOOL/></type>
        <documentation>bit 1 Etat</documentation>
</variable>
<variable name="sbDO2">
        <type><BOOL/></type>
        <documentation>bit 2 Etat</documentation>
</variable>
<variable name="sbDO3">
        <type><BOOL/></type>
        <documentation>bit 3 Etat</documentation>
</variable>
<variable name="sbDO4">
        <type><BOOL/></type>
        <documentation>bit 4 Etat</documentation>
</variable>
<variable name="sbDefaut">
        <type><BOOL/></type>
</variable>
<variable name="sbDO6">
        <type><BOOL/></type>
        <documentation>bit 6 Etat</documentation>
</variable>
<variable name="sbDO7">
        <type><BOOL/></type>
        <documentation>bit 7 Etat</documentation>
</variable>
<variable name="sbPosAtteinte">
        <type><BOOL/></type>
</variable>
<variable name="sbAxeRef">
        <type><BOOL/></type>
</variable>
<variable name="sbDO10">
        <type><BOOL/></type>
        <documentation>bit 10 Etat</documentation>
</variable>
<variable name="sbDO11">
        <type><BOOL/></type>
        <documentation>bit 11 Etat</documentation>
</variable>
<variable name="sbDO12">
        <type><BOOL/></type>
        <documentation>bit 12 Etat</documentation>
</variable>
<variable name="sbDO13">
        <type><BOOL/></type>
        <documentation>bit 13 Etat</documentation>
</variable>
<variable name="sbDO14">
        <type><BOOL/></type>
        <documentation>bit 14 Etat</documentation>
</variable>
<variable name="sbDO15">
```

```
                    <type><BOOL/></type>
                    <documentation>bit 15 Etat</documentation>
                </variable>
                <variable name="sbDO16">
                    <type><BOOL/></type>
                    <documentation>bit 16 Etat</documentation>
                </variable>
        </outputVars>
 </interface>
 <body>
        <ST>
(*conversion de EP2 + EP3 et décomposition de EP1
variables de sorties accessibles par le programme*)
siEP1:=eiEP1;
siEP2:=int_to_dint(eiEP2);
siEP3:=int_to_dint(eiEP3);
sbDO1:=eiEP1.0;
sbDO2:=eiEP1.1;
sbDO3:=eiEP1.2;
sbDO4:=eiEP1.3;
sbDefaut:=eiEP1.4;
sbDO6:=eiEP1.5;
sbDO7:=eiEP1.6;
sbPosAtteinte:=eiEP1.7;
sbAxeRef:=eiEP1.8;
sbDO10:=eiEP1.9;
sbDO11:=eiEP1.10;
sbDO12:=eiEP1.11;
sbDO13:=eiEP1.12;
sbDO14:=eiEP1.13;
sbDO15:=eiEP1.14;
sbDO16:=eiEP1.15;
        </ST>
 </body>
</pou>
```

Bloc Postérieur :

```
<pou name="nConvoyeurS" pouType="functionBlock">
 <interface>
        <inputVars>
                <variable name="eiSP2" >
                    <type><INT/></type>
                    <documentation>Donnée Physique 2</documentation>
                </variable>
                <variable name="eiSP3" >
                    <type><INT/></type>
                    <documentation>Donnée Physique 3</documentation>
                </variable>
                <variable name="ebTransfert">
```

```
                    <type><BOOL/></type>
                    <documentation>bit Transfert</documentation>
                </variable>
                <variable name="eiNbData" >
                    <type><INT/></type>
                    <documentation>bit Nombre Données Physiques pour
Transfert</documentation>
                </variable>
                <variable name="eiDataTrans1" >
                    <type><INT/></type>
                    <documentation>Donnée de Transfert 1</documentation>
                </variable>
                <variable name="eiDataTrans2" >
                    <type><INT/></type>
                    <documentation>Donnée de Transfert 2</documentation>
                </variable>
                <variable name="eiDataTrans3" >
                    <type><INT/></type>
                    <documentation>Donnée de Transfert 3</documentation>
                </variable>
                <variable name="eiDataTrans4" >
                    <type><INT/></type>
                    <documentation>Donnée de Transfert 4</documentation>
                </variable>
                <variable name="ebCmdAvance">
                    <type><BOOL/></type>
                    <documentation>bit Avance</documentation>
                </variable>
                <variable name="ebCmdRecul">
                    <type><BOOL/></type>
                    <documentation>bit Recul</documentation>
                </variable>
                <variable name="ebDI15" >
                    <type><BOOL/></type>
                    <documentation>bit 15 Etat</documentation>
                </variable>
                <variable name="ebDI16" >
                    <type><BOOL/></type>
                    <documentation>bit 16 Etat</documentation>
                </variable>
                <variable name="ebDI17" >
                    <type><BOOL/></type>
                    <documentation>bit 17 Etat</documentation>
                </variable>
                <variable name="ebJeu1" >
                    <type><BOOL/></type>
                    <documentation>bit Jeu</documentation>
                </variable>
                <variable name="ebMA">
                    <type><BOOL/></type>
                    <documentation>bit Marche/Arrêt</documentation>
                </variable>
                <variable name="ebMARapide" >
```

```
                    <type><BOOL/></type>
                    <documentation>bit Marche/Arrêt Rapide</documentation>
                </variable>
                <variable name="ebRampe1" >
                    <type><BOOL/></type>
                    <documentation>bit Rampe</documentation>
                </variable>
                <variable name="ebRecalage" >
                    <type><BOOL/></type>
                    <documentation>bit Recalage</documentation>
                </variable>
                <variable name="ebReserve" >
                    <type><BOOL/></type>
                    <documentation>Bit Reservé</documentation>
                </variable>
                <variable name="ebResetDefaut" >
                    <type><BOOL/></type>
                    <documentation>bit Reset</documentation>
                </variable>
                <variable name="ebStart" >
                    <type><BOOL/></type>
                    <documentation>bit Start</documentation>
                </variable>
                <variable name="ebVerr" >
                    <type><BOOL/></type>
                    <documentation>bit Verrouillage</documentation>
                </variable>
        </inputVars>
        <outputVars>
                <variable name="siSP1" >
                    <type><INT/></type>
                    <documentation>Sortie Commande</documentation>
                </variable>
                <variable name="siSP2" >
                    <type><INT/></type>
                    <documentation>Sortie Physique 2</documentation>
                </variable>
                <variable name="siSP3" >
                    <type><INT/></type>
                    <documentation>Sortie Physique 3</documentation>
                </variable>
        </outputVars>
        <localVars>
                <variable name="_bVerr">
                    <type><BOOL/></type>
                </variable>
                <variable name="_bMARapide"
                    <type><BOOL/></type>
                </variable>
                <variable name="_bMA"
                    <type><BOOL/></type>
                </variable>
```

```
                <variable name="_bValStart"
                        <type><BOOL/></type>
                </variable>
                <variable name="_bRampe1"
                        <type><BOOL/></type>
                </variable>
                <variable name="_bJeu1"
                        <type><BOOL/></type>
                </variable>
                <variable name="_bResetDefaut"
                        <type><BOOL/></type>
                </variable>
                <variable name="_bReserve"
                        <type><BOOL/></type>
                </variable>
                <variable name="_bStart"
                        <type><BOOL/></type>
                </variable>
                <variable name="_bCmdAvance"
                        <type><BOOL/></type>
                </variable>
                <variable name="_bCmdRecul"
                        <type><BOOL/></type>
                </variable>
                <variable name="_bRecalage"
                        <type><BOOL/></type>
                </variable>
                <variable name="_bTransfert"
                        <type><BOOL/></type>
                </variable>
                <variable name="_bDI14"
                        <type><BOOL/></type>
                </variable>
                <variable name="_bDI15"
                        <type><BOOL/></type>
                </variable>
                <variable name="_bDI16"
                        <type><BOOL/></type>
                </variable>
                <variable name="_iSP2"
                        <type><INT/></type>
                </variable>
                <variable name="_iSP3"
                        <type><INT/></type>
                </variable>
        </localVars>
 </interface>
 <body>
        <ST>
(*mise à jour des sorties Physiques
à partir des variables publiques*)
siSP1.0:=_bVerr;
siSP1.1:=_bMARapide;
```

```
siSP1.2:=_bMA;
siSP1.3:=_bValStart;
siSP1.4:=_bRampe1;
siSP1.5:=_bJeu1;
siSP1.6:=_bResetDefaut;
siSP1.7:=_bReserve;
siSP1.8:=_bStart;
siSP1.9:=_bCmdAvance;
siSP1.10:=_bCmdRecul;
siSP1.11:=_bRecalage;
siSP1.12:=_bTransfert;
siSP1.13:=_bDI14;
siSP1.14:=_bDI15;
siSP1.15:=_bDI16;
siSP2:=_iSP2;
siSP3:=_iSP3;
                </ST>
  </body>
</pou>
```

A ce stade, nous avons toutes les définitions nécessaires pour caractériser notre bloc fonction et nous n'avons pas encore déterminé la plate-forme cible et notamment son environnement de développement (*IDE*). C'est l'avantage de cette approche, repousser la programmation dans la technologie cible le plus tard possible.

Nous choisissons dans notre cas d'étude l'environnement *Unity* pour intégrer la spécification de nos blocs. Unity possède ses propres schémas de structure pour la description des blocs fonction en **XML**. La conversion se réalise facilement entre le modèle PLCopen et le modèle Unity pour la définition d'un bloc, à cela près qu'il faut incrémenter un index pour chaque nouvelle entrée ou sortie détectée (attribut « *PositionPin* » avec « *value* »). Les fichiers obtenus sont enregistrés en ***.XDB**. (stockage des définition de blocs fonctions sous unity)

Bloc Préliminaire :

```
<FBSource nameOfFBType="nConvoyeurE" version="0.16">
        <comment>bloc de commande entrée Brushless 3dp</comment>
        <inputParameters>
                <variables name="eiEP1" typeName="INT">
                        <comment>Retour Etat</comment>
                        <attribute name="PositionPin" value="1"></attribute>
                </variables>
                <variables name="eiEP2" typeName="INT">
                        <comment>Retour Donnée Physique 2</comment>
                        <attribute name="PositionPin" value="2"></attribute>
```

```
            </variables>
            <variables name="eiEP3" typeName="INT">
                <comment>Retour Donnée Physique 3</comment>
                <attribute name="PositionPin" value="3"></attribute>
            </variables>
        </inputParameters>
        <outputParameters>
            <variables name="siEP1" typeName="INT">
                <comment>Etat</comment>
                <attribute name="PositionPin" value="1"></attribute>
            </variables>
            <variables name="siEP2" typeName="DINT">
                <comment>Données Physique 2</comment>
                <attribute name="PositionPin" value="2"></attribute>
            </variables>
            <variables name="siEP3" typeName="DINT">
                <comment>Données Physique 2</comment>
                <attribute name="PositionPin" value="3"></attribute>
            </variables>
            <variables name="sbDO1" typeName="BOOL">
                <comment>bit 1 Etat</comment>
                <attribute name="PositionPin" value="4"></attribute>
            </variables>
            <variables name="sbDO2" typeName="BOOL">
                <comment>bit 2 Etat</comment>
                <attribute name="PositionPin" value="5"></attribute>
            </variables>
            <variables name="sbDO3" typeName="BOOL">
                <comment>bit 3 Etat</comment>
                <attribute name="PositionPin" value="6"></attribute>
            </variables>
            <variables name="sbDO4" typeName="BOOL">
                <comment>bit 4 Etat</comment>
                <attribute name="PositionPin" value="7"></attribute>
            </variables>
            <variables name="sbDefaut" typeName="BOOL">
                <attribute name="PositionPin" value="8"></attribute>
            </variables>
            <variables name="sbDO6" typeName="BOOL">
                <comment>bit 6 Etat</comment>
                <attribute name="PositionPin" value="9"></attribute>
            </variables>
            <variables name="sbDO7" typeName="BOOL">
                <comment>bit 7 Etat</comment>
                <attribute name="PositionPin" value="10"></attribute>
            </variables>
            <variables name="sbPosAtteinte" typeName="BOOL">
                <attribute name="PositionPin" value="11"></attribute>
            </variables>
            <variables name="sbAxeRef" typeName="BOOL">
                <attribute name="PositionPin" value="12"></attribute>
            </variables>
            <variables name="sbDO10" typeName="BOOL">
```

```
                <comment>bit 10 Etat</comment>
                <attribute name="PositionPin" value="13"></attribute>
            </variables>
            <variables name="sbDO11" typeName="BOOL">
                <comment>bit 11 Etat</comment>
                <attribute name="PositionPin" value="14"></attribute>
            </variables>
            <variables name="sbDO12" typeName="BOOL">
                <comment>bit 12 Etat</comment>
                <attribute name="PositionPin" value="15"></attribute>
            </variables>
            <variables name="sbDO13" typeName="BOOL">
                <comment>bit 13 Etat</comment>
                <attribute name="PositionPin" value="16"></attribute>
            </variables>
            <variables name="sbDO14" typeName="BOOL">
                <comment>bit 14 Etat</comment>
                <attribute name="PositionPin" value="17"></attribute>
            </variables>
            <variables name="sbDO15" typeName="BOOL">
                <comment>bit 15 Etat</comment>
                <attribute name="PositionPin" value="18"></attribute>
            </variables>
            <variables name="sbDO16" typeName="BOOL">
                <comment>bit 16 Etat</comment>
                <attribute name="PositionPin" value="19"></attribute>
            </variables>
    </outputParameters>
    <FBProgram name="SEC1">
    <STSource>(*conversion de EP2 + EP3 et décomposition de EP1
            variables de sorties accessibles par le programme*)
            siEP1:=eiEP1;
            siEP2:=int_to_dint(eiEP2);
            siEP3:=int_to_dint(eiEP3);
            sbDO1:=eiEP1.0;
            sbDO2:=eiEP1.1;
            sbDO3:=eiEP1.2;
            sbDO4:=eiEP1.3;
            sbDefaut:=eiEP1.4;
            sbDO6:=eiEP1.5;
            sbDO7:=eiEP1.6;
            sbPosAtteinte:=eiEP1.7;
            sbAxeRef:=eiEP1.8;
            sbDO10:=eiEP1.9;
            sbDO11:=eiEP1.10;
            sbDO12:=eiEP1.11;
            sbDO13:=eiEP1.12;
            sbDO14:=eiEP1.13;
            sbDO15:=eiEP1.14;
            sbDO16:=eiEP1.15;
            </STSource>
    </FBProgram>
```

```
</FBSource>
```

Bloc Postérieur :

```
<FBSource nameOfFBType="nConvoyeurS" version="0.06">
        <inputParameters>
                <variables name="eiSP2" typeName="INT">
                        <comment>Donnée Physique 2</comment>
                        <attribute name="PositionPin" value="22"></attribute>
                </variables>
                <variables name="eiSP3" typeName="INT">
                        <comment>Donnée Physique 3</comment>
                        <attribute name="PositionPin" value="23"></attribute>
                </variables>
                <variables name="ebTransfert" typeName="BOOL">
                        <comment>bit Transfert</comment>
                        <attribute name="PositionPin" value="15"></attribute>
                </variables>
                <variables name="eiNbData" typeName="INT">
                        <comment>bit Nombre Données Physiques pour Transfert</comment>
                        <attribute name="PositionPin" value="16"></attribute>
                </variables>
                <variables name="eiDataTrans1" typeName="INT">
                        <comment>Donnée de Transfert 1</comment>
                        <attribute name="PositionPin" value="17"></attribute>
                </variables>
                <variables name="eiDataTrans2" typeName="INT">
                        <comment>Donnée de Transfert 2</comment>
                        <attribute name="PositionPin" value="18"></attribute>
                </variables>
                <variables name="eiDataTrans3" typeName="INT">
                        <comment>Donnée de Transfert 3</comment>
                        <attribute name="PositionPin" value="19"></attribute>
                </variables>
                <variables name="eiDataTrans4" typeName="INT">
                        <comment>Donnée de Transfert 4</comment>
                        <attribute name="PositionPin" value="20"></attribute>
                </variables>
                <variables name="ebCmdAvance" typeName="BOOL">
                        <comment>bit Avance</comment>
                        <attribute name="PositionPin" value="1"></attribute>
                </variables>
                <variables name="ebCmdRecul" typeName="BOOL">
                        <comment>bit Recul</comment>
                        <attribute name="PositionPin" value="2"></attribute>
                </variables>
                <variables name="ebDI15" typeName="BOOL">
                        <comment>bit 15 Etat</comment>
                        <attribute name="PositionPin" value="3"></attribute>
                </variables>
                <variables name="ebDI16" typeName="BOOL">
                        <comment>bit 16 Etat</comment>
                        <attribute name="PositionPin" value="4"></attribute>
```

```
            </variables>
            <variables name="ebDI17" typeName="BOOL">
                <comment>bit 17 Etat</comment>
                <attribute name="PositionPin" value="5"></attribute>
            </variables>
            <variables name="ebJeu1" typeName="BOOL">
                <comment>bit Jeu</comment>
                <attribute name="PositionPin" value="6"></attribute>
            </variables>
            <variables name="ebMA" typeName="BOOL">
                <comment>bit Marche/Arrêt</comment>
                <attribute name="PositionPin" value="7"></attribute>
            </variables>
            <variables name="ebMARapide" typeName="BOOL">
                <comment>bit Marche/Arrêt Rapide</comment>
                <attribute name="PositionPin" value="8"></attribute>
            </variables>
            <variables name="ebRampe1" typeName="BOOL">
                <comment>bit Rampe</comment>
                <attribute name="PositionPin" value="9"></attribute>
            </variables>
            <variables name="ebRecalage" typeName="BOOL">
                <comment>bit Recalage</comment>
                <attribute name="PositionPin" value="10"></attribute>
            </variables>
            <variables name="ebReserve" typeName="BOOL">
                <comment>Bit Reservé</comment>
                <attribute name="PositionPin" value="11"></attribute>
            </variables>
            <variables name="ebResetDefaut" typeName="BOOL">
                <comment>bit Reset</comment>
                <attribute name="PositionPin" value="12"></attribute>
            </variables>
            <variables name="ebStart" typeName="BOOL">
                <comment>bit Start</comment>
                <attribute name="PositionPin" value="13"></attribute>
            </variables>
            <variables name="ebVerr" typeName="BOOL">
                <comment>bit Verrouillage</comment>
                <attribute name="PositionPin" value="14"></attribute>
            </variables>
        </inputParameters>
        <outputParameters>
            <variables name="siSP1" typeName="INT">
                <comment>Sortie Commande</comment>
                <attribute name="PositionPin" value="1"></attribute>
            </variables>
            <variables name="siSP2" typeName="INT">
                <comment>Sortie Physique 2</comment>
                <attribute name="PositionPin" value="2"></attribute>
            </variables>
            <variables name="siSP3" typeName="INT">
```

```
                        <comment>Sortie Physique 3</comment>
                        <attribute name="PositionPin" value="3"></attribute>
                </variables>
        </outputParameters>
        <publicLocalVariables>
                <variables name="_bVerr" typeName="BOOL"></variables>
                <variables name="_bMARapide" typeName="BOOL"></variables>
                <variables name="_bMA" typeName="BOOL"></variables>
                <variables name="_bValStart" typeName="BOOL"></variables>
                <variables name="_bRampe1" typeName="BOOL"></variables>
                <variables name="_bJeu1" typeName="BOOL"></variables>
                <variables name="_bResetDefaut" typeName="BOOL"></variables>
                <variables name="_bReserve" typeName="BOOL"></variables>
                <variables name="_bStart" typeName="BOOL"></variables>
                <variables name="_bCmdAvance" typeName="BOOL"></variables>
                <variables name="_bCmdRecul" typeName="BOOL"></variables>
                <variables name="_bRecalage" typeName="BOOL"></variables>
                <variables name="_bTransfert" typeName="BOOL"></variables>
                <variables name="_bDI14" typeName="BOOL"></variables>
                <variables name="_bDI15" typeName="BOOL"></variables>
                <variables name="_bDI16" typeName="BOOL"></variables>
                <variables name="_iSP2" typeName="INT"></variables>
                <variables name="_iSP3" typeName="INT"></variables>
        </publicLocalVariables>
        <FBProgram name="SEC1">
                <STSource>(*mise à jour des sorties Physiques
à partir des variables publiques*)
siSP1.0:=_bVerr;
siSP1.1:=_bMARapide;
siSP1.2:=_bMA;
siSP1.3:=_bValStart;
siSP1.4:=_bRampe1;
siSP1.5:=_bJeu1;
siSP1.6:=_bResetDefaut;
siSP1.7:=_bReserve;
siSP1.8:=_bStart;
siSP1.9:=_bCmdAvance;
siSP1.10:=_bCmdRecul;
siSP1.11:=_bRecalage;
siSP1.12:=_bTransfert;
siSP1.13:=_bDI14;
siSP1.14:=_bDI15;
siSP1.15:=_bDI16;
siSP2:=_iSP2;
siSP3:=_iSP3;
                </STSource>
        </FBProgram>
</FBSource>
```

Ensuite, après ouverture d'Unity dans n'importe quelle configuration, il suffit d'aller dans l'arborescence des données, d'effectuer un clic droit sur « *Types FB dérivés* »

puis d'importer le fichier *.**XDB** :

On retrouve les blocs importés et correspondant à *nConvoyeur* sous cette forme dans la fenêtre des Types DFB :

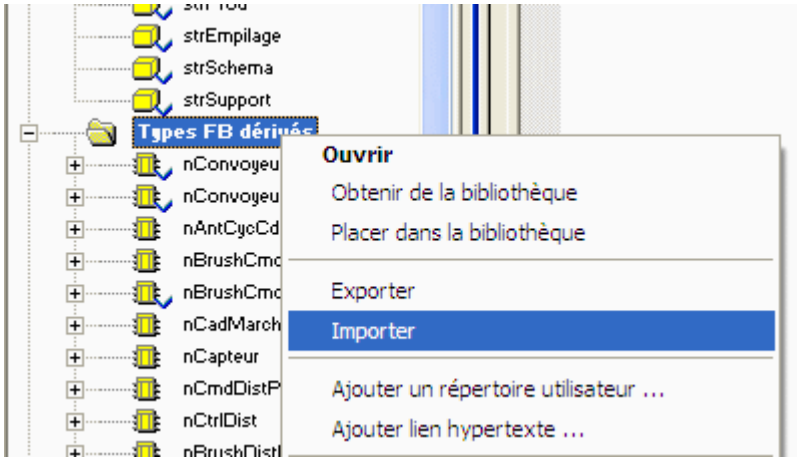


Figure 9.11: Importation des blocs sous Unity

nConvoyeurE		<DFB>		bloc de commande entrée Brushless 3dp
<entrées>				
eiEP1	1	INT		Retour Etat
eiEP2	2	INT		Retour Donnée Physique 2
eiEP3	3	INT		Retour Donnée Physique 3
<sorties>				
siEP1	1	INT		Etat
siEP2	2	DINT		Données Physique 2
siEP3	3	DINT		Données Physique 2
sbDO1	4	BOOL		bit 1 Etat
sbDO2	5	BOOL		bit 2 Etat
sbDO3	6	BOOL		bit 3 Etat
sbDO4	7	BOOL		bit 4 Etat
sbDefaut	8	BOOL		
sbDO6	9	BOOL		bit 6 Etat
sbDO7	10	BOOL		bit 7 Etat
sbPosAtteinte	11	BOOL		
sbAxeRef	12	BOOL		
sbDO10	13	BOOL		bit 10 Etat
sbDO11	14	BOOL		bit 11 Etat
sbDO12	15	BOOL		bit 12 Etat
sbDO13	16	BOOL		bit 13 Etat
sbDO14	17	BOOL		bit 14 Etat
sbDO15	18	BOOL		bit 15 Etat
sbDO16	19	BOOL		bit 16 Etat

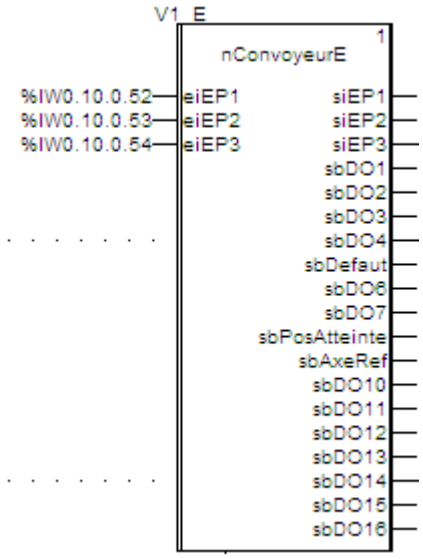
Figure 9.12: Définition du bloc fonction nConvoyeurE

Bloc Postérieur :

nConvoyeurS		<DFB>		
<entrées>				
ebCmdAvance	1	BOOL		bit Avance
ebCmdRecul	2	BOOL		bit Recul
ebDI15	3	BOOL		bit 15 Etat
ebDI16	4	BOOL		bit 16 Etat
ebDI17	5	BOOL		bit 17 Etat
ebJeu1	6	BOOL		bit Jeu
ebMA	7	BOOL		bit Marche/Arrêt
ebMARapide	8	BOOL		bit Marche/Arrêt Rapide
ebRampe1	9	BOOL		bit Rampe
ebRecalage	10	BOOL		bit Recalage
ebReserve	11	BOOL		Bit Reservé
ebResetDefaut	12	BOOL		bit Reset
ebStart	13	BOOL		bit Start
ebVerr	14	BOOL		bit Verrouillage
ebTransfert	15	BOOL		bit Transfert
eiNbData	16	INT		bit Nombre Données Physiques pour Transfert
eiDataTrans1	17	INT		Donnée de Transfert 1
eiDataTrans2	18	INT		Donnée de Transfert 2
eiDataTrans3	19	INT		Donnée de Transfert 3
eiDataTrans4	20	INT		Donnée de Transfert 4
eiSP2	22	INT		Donnée Physique 2
eiSP3	23	INT		Donnée Physique 3
<sorties>				
siSP1	1	INT		Sortie Commande
siSP2	2	INT		Sortie Physique 2
siSP3	3	INT		Sortie Physique 3

Figure 9.13: Définition du bloc fonction nConvoyeurS

Exemple d'utilisation comme instances dans le programme de la préparation robotisée :



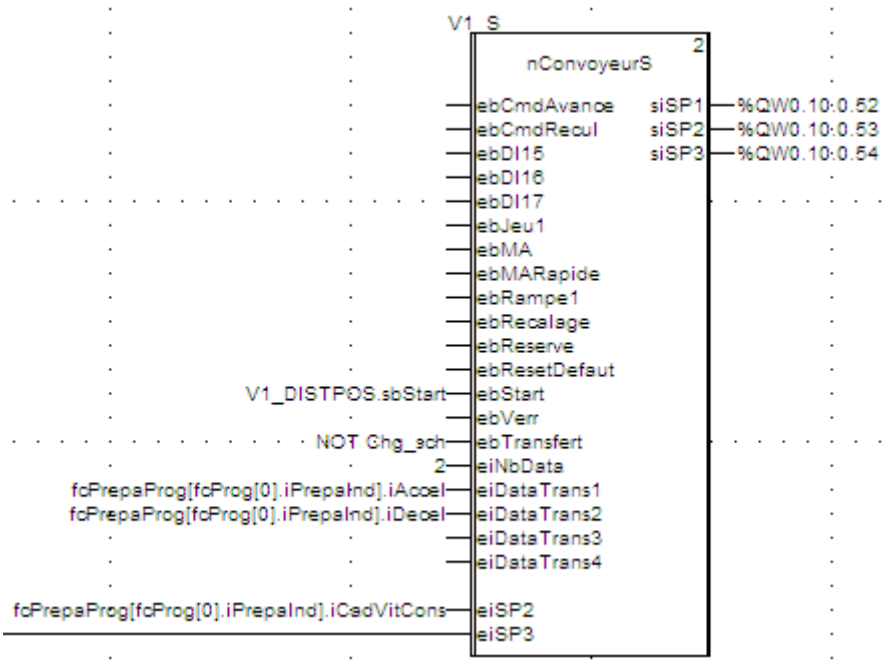


Figure 9.14: Instances des blocs nConvoyeur E et nConvoyeurS sous Unity

Nous présentons ensuite les autres briques élémentaires sans répéter les explications liées aux langages et aux formats d'exportation/conversion.

9.2.3 Brique élémentaire nCapteur

9.2.3.1 Facette Physique

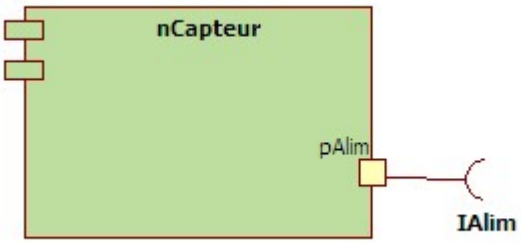


Figure 9.15: Vue Physique nCapteur

Le capteur de proximité utilisé a simplement besoin d'une alimentation électrique. Et nous ne considérons pas ses éléments internes (charge du fournisseur).

9.2.3.2 Facette Interface Utilisateur

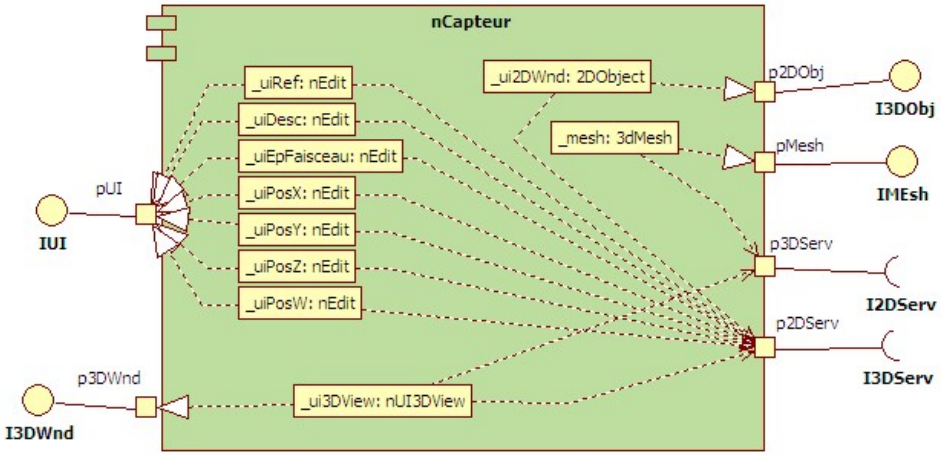


Figure 9.16: Vue Interface Utilisateur de nCapteur

De la même façon que pour le convoyeur, on retrouve les éléments graphiques permettant à l'utilisateur de renseigner ou de visualiser des informations sur le capteur.

9.2.3.3 Facette Logique

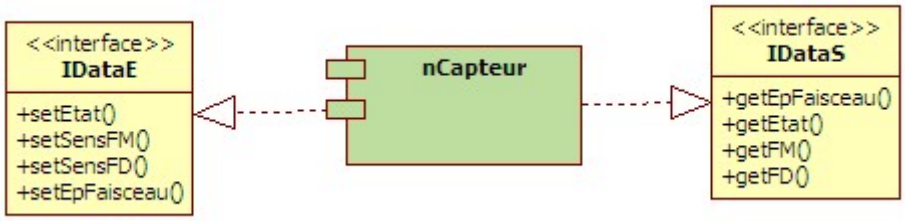


Figure 9.17: Vue boîte noire de l'objet nCapteur

Comme pour l'élément *nConvoyeur*, on retrouve une interface d'entrée *IDataE*. *SetEtat()* permet de renseigner l'objet sur l'état physique du capteur réel. Deux autres fonctions *setSensFM()* et *setSensFD()* permettent de définir des sensibilités de détection afin de filtrer les fronts montants ou descendants du signal. Ainsi un changement d'état du signal dont la durée est inférieure à la valeur fixée par la sensibilité ne sera pas pris en compte dans le programme. *SetEpFaisceau()* renseigne l'objet sur l'épaisseur du faisceau de détection du capteur afin que le programme puisse éventuellement apporter des correctifs sur les éléments détectés (longueur mesurée par exemple).

En sortie de l'objet on retrouve notamment la possibilité de lire les fronts montants (FM) et les fronts descendants (FD) du signal, l'épaisseur faisceau et l'état courant du signal. Les autres objet peuvent donc se servir de ces informations déjà filtrées pour la gestion de leur comportement. Sur la figure 9.18 on retrouve la vue SysML de l'objet précédent.

Figure 9.18: Vue SysML de nCapteur

Avec les mêmes étapes que dans le cas du convoyeur, en passant par une première génération en XMI, puis en convertissant successivement en XML PLCopen et XML Unity Pro, nous obtenons le fichier suivant :

```
<FBSource nameOfFBType="nCapteur" version="0.09">
        <comment>bloc capteur</comment>
        <inputParameters>
                <variables name="ebEtat" typeName="BOOL">
                        <comment>signal d'entrée</comment>
                        <attribute name="PositionPin" value="1"></attribute>
                </variables>
                <variables name="eiSensFM" typeName="INT">
                        <comment>sensibilité Front Montant en ms</comment>
                        <attribute name="PositionPin" value="2"></attribute>
                </variables>
                <variables name="eiSensFD" typeName="INT">
                        <comment>sensibilité Front Descendant en ms</comment>
                        <attribute name="PositionPin" value="3"></attribute>
                </variables>
                <variables name="eiEpFaisceau" typeName="INT">
                        <attribute name="PositionPin" value="4"></attribute>
                </variables>
        </inputParameters>
        <outputParameters>
                <variables name="sbEtat" typeName="BOOL">
                        <comment>signal filtré</comment>
                        <attribute name="PositionPin" value="1"></attribute>
                </variables>
                <variables name="sbFM" typeName="BOOL">
                        <comment>front montant</comment>
                        <attribute name="PositionPin" value="2"></attribute>
                </variables>
                <variables name="sbFD" typeName="BOOL">
                        <comment>front descendant</comment>
                        <attribute name="PositionPin" value="3"></attribute>
                </variables>
                <variables name="siEpFaisceau" typeName="INT">
                        <comment>épaisseur faisceau</comment>
```

```
                    <attribute name="PositionPin" value="4"></attribute>
                </variables>
            </outputParameters>
            <privateLocalVariables>
                <variables name="_tempoFM" typeName="TON"></variables>
                <variables name="_front" typeName="TRIGGER"></variables>
                <variables name="_tempoFD" typeName="TON"></variables>
            </privateLocalVariables>
            <FBProgram name="SEC1">
                <STSource>(*temporisation*)
                    _tempoFM(IN:=ebEtat, PT:=INT_TO_TIME(eiSensFM));
                    _tempoFD(IN:=NOT ebEtat, PT:=INT_TO_TIME(eiSensFD));

                    (*mis à jour état*)
                    if(sbEtat AND _tempoFD.Q) then sbEtat:=0;
                    elsif(NOT sbEtat AND _tempoFM.Q) then sbEtat:=1;
                    end_if;
                    (*détection des fronts*)
                    _front (CLK:=sbEtat);

                    (*sorties*)
                    sbFM:=_front.RISE;
                    sbFD:=_front.FALL;
                    siEpFaisceau:=eiEpFaisceau;
                </STSource>
            </FBProgram>
</FBSource>
```

Le code ST a été renseigné directement dans l'environnement de développement cible. Il n'y a pas pour le moment de structure XML standardisée par PLCopen et permettant de décrire les langages textuels (Structuré et LIST). Néanmoins, nous verrons plus loin qu'il est possible, à l'aide d'un diagramme d'activité SysML lié au block, de spécifier le comportement dynamique interne de l'objet et donc d'établir une conception de la logique du bloc fonction hors d'un environnement de développement spécifique.

La brique élémentaire suivante représente un robot 4 axes, nous reprenons à nouveau les mêmes étapes de spécification.

9.2.4 Brique élémentaire nRobot4A

9.2.4.1 Facette Physique

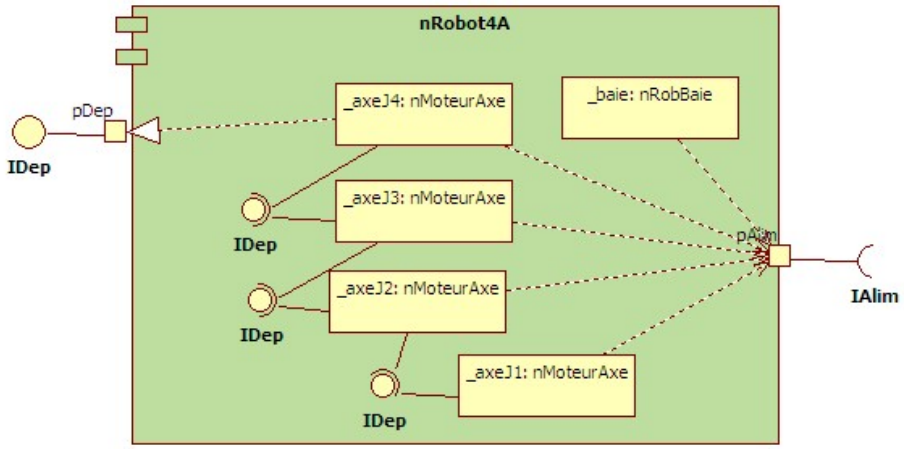


Figure 9.19: Vue Physique de nRobot4A

On retrouve en interne 4 instances de *nMoteurAxe* pour caractériser les 4 axes du robot. On a également la présence d'une baie robot permettant l'exécution du programme utilisateur et la gestion des mouvements du robot. Le robot doit être alimenté en électricité via *IAlim* et le poignet du robot situé sur l'axe J4 exerce le mouvement résultant de la combinaison des rotations (caractérisées par des interfaces de déplacement *IDep*) de chaque axe.

9.2.4.2 Facette interface utilisateur

Comme pour les briques élémentaires précédentes, nous définissons ici l'interface de communication entre l'utilisateur et le système de gestion. On retrouve des champs d'édition pour le paramétrage de l'objet, des fenêtres de visualisation et des objets graphiques réutilisables pour des affichages externes 2D ou 3D. La figure 9.20 montre la modélisation effectuée à ce niveau.

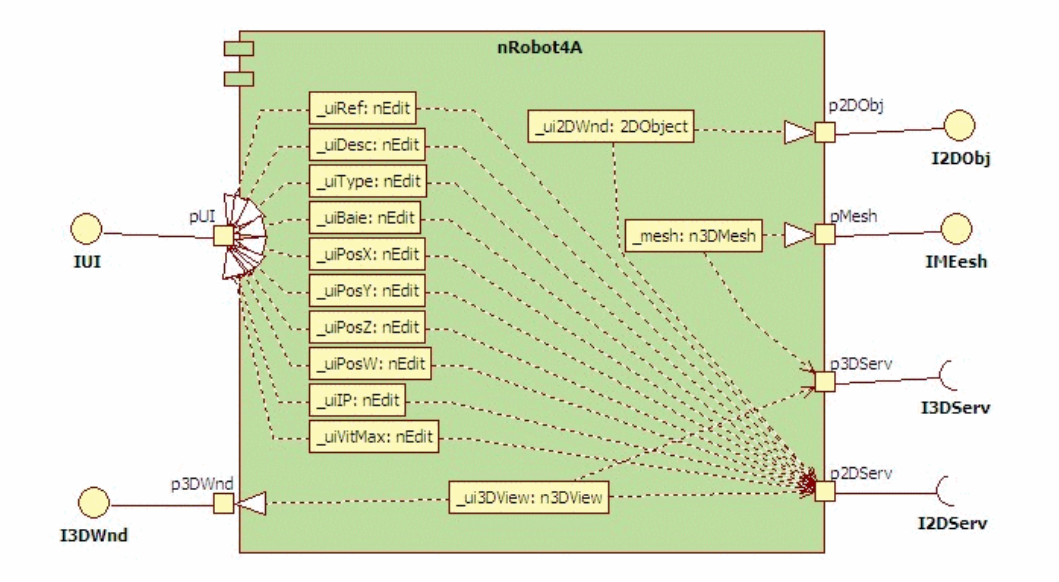


Figure 9.20: Vue interface utilisateur de nRobot4A

9.2.4.3 Facette Logique

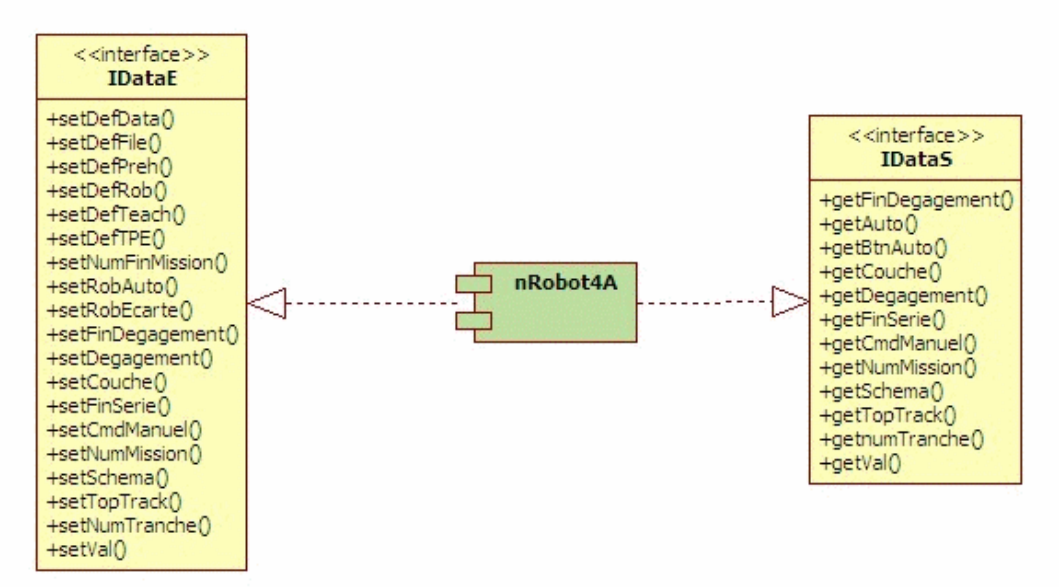


Figure 9.21: Vue logique nRobot4A

Sur l'interface *IDataE* on retrouve l'ensemble des services qui permettent de donner des informations à l'objet *nRobot4A* pour son traitement interne : la détection de défauts (données, fichier, préhenseur, Baie Robot, *Teach Pendan*t, programme TPE), les missions à réaliser, les cycles à effectuer (Auto, Manuel, fin de série, écartement, dégagement, *tracking*), les données de palettisation (schéma, couche).

L'interface IDataS met à disposition des autres objets des informations concernant son cycle courant, les données de palettisation actuelles et le numéro de mission.

Sur la figure 9.22 on retrouve l'expression de ce diagramme en SysML.

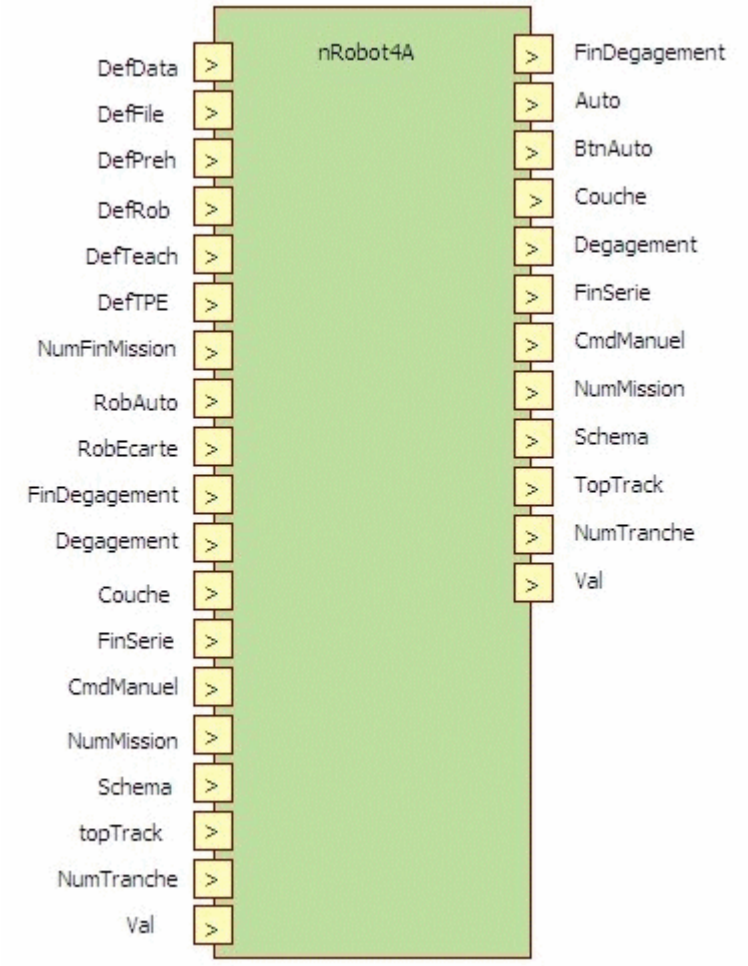


Figure 9.22: Diagramme SysML de l'objet nRobot4A

De la même façon que pour le convoyeur, après avoir choisi comme cible l'environnement Unity Pro, nous décidons de séparer l'objet en deux blocs fonctions, pour permettre au programme de traiter certaines données avant l'écriture des sorties physiques. Les fichiers XML Unity obtenus sont présentés ci-après.

Bloc préliminaire :

```
<FBSource nameOfFBType="nPrepaRobotE" version="0.04" >
        <inputParameters>
                <variables name="ebDefData" typeName="BOOL">
                        <attribute name="PositionPin" value="1"></attribute>
                </variables>
                <variables name="ebDefFile" typeName="BOOL">
                        <attribute name="PositionPin" value="2"></attribute>
                </variables>
                <variables name="ebDefPreh" typeName="BOOL">
                        <attribute name="PositionPin" value="3"></attribute>
                </variables>
                <variables name="ebDefRob" typeName="BOOL">
                        <attribute name="PositionPin" value="4"></attribute>
                </variables>
                <variables name="ebDefTeach" typeName="BOOL">
                        <attribute name="PositionPin" value="5"></attribute>
                </variables>
                <variables name="ebDefTPE" typeName="INT">
                        <attribute name="PositionPin" value="6"></attribute>
                </variables>
                <variables name="eiNumFinMission" typeName="INT">
                        <attribute name="PositionPin" value="7"></attribute>
                </variables>
                <variables name="ebRobAuto" typeName="BOOL">
                        <attribute name="PositionPin" value="8"></attribute>
                </variables>
                <variables name="ebRobEcarte" typeName="BOOL">
                        <attribute name="PositionPin" value="9"></attribute>
                </variables>
        </inputParameters>
        <FBProgram name="SEC1">
                [.....]
        </FBProgram>
</FBSource>
```

Bloc postérieur :

```
<FBSource nameOfFBType="nPrepaRobotS" version="0.09">
        <inputParameters>
                <variables name="ebFinDegagement" typeName="BOOL">
                        <attribute name="PositionPin" value="1"></attribute>
```

```
            </variables>
            <variables name="ebAuto" typeName="BOOL">
                  <attribute name="PositionPin" value="2"></attribute>
            </variables>
            <variables name="ebBtnAuto" typeName="BOOL">
                  <attribute name="PositionPin" value="3"></attribute>
            </variables>
            <variables name="eiCouche" typeName="INT">
                  <attribute name="PositionPin" value="4"></attribute>
            </variables>
            <variables name="ebDegagement" typeName="BOOL">
                  <attribute name="PositionPin" value="5"></attribute>
            </variables>
            <variables name="ebFinSerie" typeName="BOOL">
                  <attribute name="PositionPin" value="6"></attribute>
            </variables>
            <variables name="eiCmdManuel" typeName="INT">
                  <attribute name="PositionPin" value="7"></attribute>
            </variables>
            <variables name="eiNumMission" typeName="INT">
                  <attribute name="PositionPin" value="8"></attribute>
            </variables>
            <variables name="eiSchema" typeName="INT">
                  <attribute name="PositionPin" value="9"></attribute>
            </variables>
            <variables name="ebTopTrack" typeName="BOOL">
                  <attribute name="PositionPin" value="10"></attribute>
            </variables>
            <variables name="eiNumTranche" typeName="INT">
                  <attribute name="PositionPin" value="11"></attribute>
            </variables>
            <variables name="ebVal" typeName="BOOL">
                  <attribute name="PositionPin" value="12"></attribute>
            </variables>
      </inputParameters>
      <outputParameters>
            <variables name="sbFinDegagement" typeName="BOOL">
                  <attribute name="PositionPin" value="1"></attribute>
            </variables>
            <variables name="sbAuto" typeName="BOOL">
                  <attribute name="PositionPin" value="2"></attribute>
            </variables>
            <variables name="sbBtnAuto" typeName="BOOL">
                  <attribute name="PositionPin" value="3"></attribute>
            </variables>
            <variables name="siCouche" typeName="INT">
                  <attribute name="PositionPin" value="4"></attribute>
            </variables>
            <variables name="sbDegagement" typeName="BOOL">
                  <attribute name="PositionPin" value="5"></attribute>
            </variables>
            <variables name="sbFinSerie" typeName="BOOL">
                  <attribute name="PositionPin" value="6"></attribute>
```

```
                </variables>
                <variables name="siCmdManuel" typeName="INT">
                        <attribute name="PositionPin" value="7"></attribute>
                </variables>
                <variables name="siNumMission" typeName="INT">
                        <attribute name="PositionPin" value="8"></attribute>
                </variables>
                <variables name="siSchema" typeName="INT">
                        <attribute name="PositionPin" value="9"></attribute>
                </variables>
                <variables name="sbTopTrack" typeName="BOOL">
                        <attribute name="PositionPin" value="10"></attribute>
                </variables>
                <variables name="siNumTranche" typeName="INT">
                        <attribute name="PositionPin" value="11"></attribute>
                </variables>
                <variables name="sbVal" typeName="BOOL">
                        <attribute name="PositionPin" value="12"></attribute>
                </variables>
        </outputParameters>
        <FBProgram name="SEC1">
                [.....................]
        </FBProgram>
</FBSource>
```

9.3 **Identification de nouvelles briques**

Les briques précédentes ont été réalisées dans le cadre de l'AOM, décrite dans la méthodologie du chapitre précédent. Lors de l'analyse d'une machine plus complexe, l'approche descendante identifie des fonctionnalités supplémentaires au niveau de la logique de contrôle notamment. Ces besoins sont exprimés auprès des personnes réalisant le standard de contrôle (AOM)

Tout d'abord, les convoyeurs sont pilotés en indiquant une destination cible à atteindre au niveau de leur position (via le codeur du moteur Brushless). Or, si l'on regarde le fonctionnement du cadencement de la préparation robotisée, on voit que ce sont des distances qui seront décrites (longueurs des groupes de produits). Il faut donc un premier objet capable de convertir des distances en consignes de position et un second capable, à l'inverse, de transformer des changements de position en distances.

De plus la gestion des cadenceurs impose un ordonnancement d'ordres de

marche, d'ordres d'arrêt sous forme de distances. Ces distances pourront être ajustées pour recaler un éventuel décalage dans la réalité. Un troisième objet permettra de gérer ces distances.

L'envoi des groupes de produits au niveau des cadenceurs doit se faire de façon à obtenir des trains de colis centrés les uns par rapport aux autres dans une même tranche, comme sur la figure 9.2. Cette contrainte impose un contrôle que nous réaliserons au niveau d'une cellule située sur le convoyeur aval de façon à mesurer d'éventuels décalages le plus précisément possible. La gestion de cette synchronisation de cadencement impose la création d'un objet de synchronisation.

Pour récapituler nous avons donc identifié quatre éléments à réaliser au niveau du standard des objets de contrôle :

- conversion de distance en position avec prise en charge du rebouclage de position.
- conversion de position en distance avec prise en charge du rebouclage de position.
- gestionnaire d'ordres de marche/arrêt en distances avec d'éventuels recalages.
- gestionnaire de synchronisation d'évènements.

Une spécification de ces objets, indépendamment du contexte de la préparation robotisée, va être réalisée par les personnes chargées de la définition des objets standards, selon le même principe que celui utilisé pour la définition des briques élémentaires présentées précédemment.

9.3.1 Conversion de distance en position

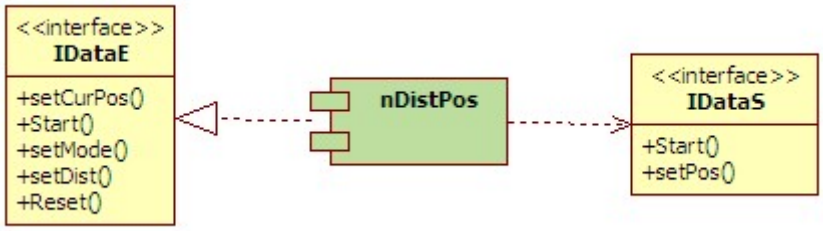


Figure 9.23: Vue Logique de l'objet nDistPos

Cet objet reçoit un ordre de contrôle via son interface IDataE et le service « *Start* ». Il récupère alors une consigne de distance via « *setDist*() », la position courante de l'élément à contrôler, puis il envoie un ordre de marche et une position de destination cible à partir de ces données.

Après les étapes de génération/conversion on peut importer l'objet individuel sous Unity :

<entrées>				
ediCurPos	1	DINT		position courante
ebStart	2	BOOL		ordre de marche
ebMode	3	BOOL		avance continue
eiDist	4	INT		consigne d'avance
ebReset	5	BOOL		reset bloc
<sorties>				
sbStart	1	BOOL		ordre de marche
sdiDestination	2	INT		destination

Figure 9.24: Définition du bloc fonction nDistPos sous Unity Pro

9.3.2 Conversion de position en distance

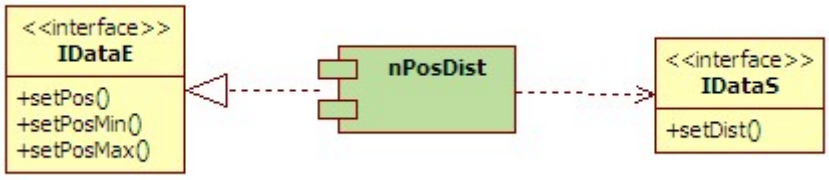


Figure 9.25: Vue logique de l'objet nPosDist

Lorsque ce bloc reçoit une nouvelle position sur son interface IDataE via le service « *setPos*() », il détermine la distance effectuée depuis sa dernière exécution. Pour tenir compte d'éventuelles limites au niveau des valeurs de position, on lui indique également une position minimale et une position maximale pour gérer les boucles de position (exemple dans le cas d'un codeur, une position de 100 avec une position précédente de 5500 donne une distance effectuée de 200 avec une position minimale à 50 et une position maximale à 5650. sans la gestion du bouclage on obtiendrait une position erronée de -5400).

<entrées>				
eDistance	1	INT		distance de consigne
ePosition	2	INT		position de l'élément à contrôler
eStart	3	EBOOL		nouvelle consigne
<sorties>				
sPosition	1	INT		position cible

Figure 9.26: Définition du bloc fonction nPosDist sous UnityPro

9.3.3 Gestionnaire de Marche/Arrêt

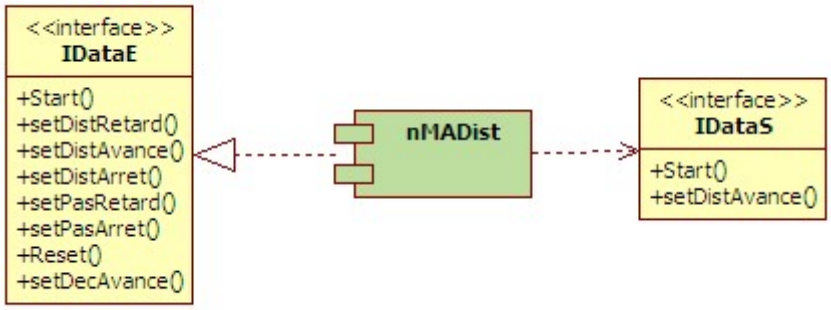


Figure 9.27: Vue Logique de l'objet nMADist

Lorsque l'interface *IDataE* reçoit un ordre de contrôle via le service « *Start*() » l'objet utilisera des valeurs mises à jour par les services « *setDistRetard*() », « *setDistAvance*() », « *setDistArret*() », « *setPasRetard*() », « *setPasArret*() », « *setDecAvance*() », pour gérer les phases de contrôle suivantes :

- Maintien d'un arrêt tant que la distance de retard de démarrage n'a pas été réalisée. Le contrôle de cette distance se base sur un autre élément externe, inconnu, et dont on ne récupère qu'un *pas,* via *setPasRetard*, permettant de mettre à jour la mémorisation d'une distance de retard déjà effectuée.

- Appel du service « *Start*() » et du service « *setDistAvance*() » lorsque la distance de retard à été effectuée. La distance d'avance commandée est réduite par la consigne indiquée via « *setDecAvance*() ».

- Lorsque la distance d'avance est effectuée, on provoque à nouveau un arrêt lié à la distance indiquée par « *setDistArret*() ». Le contrôle de cette distance se base sur un autre élément externe, inconnu dont on ne récupère qu'un *pas,* via *setPasArret*, permettant de mettre à jour la mémorisation d'une distance d'arrêt déjà effectuée.

<entrées>				
ebStart	1	BOOL		top gestion tranche
eiDistRetard	2	INT		distance de retard
eiDistAvance	3	INT		distance d'avance
eiDistArret	4	INT		distance d'arrêt
eiPasRetard	5	INT		pas pour la mesure du retard
eiPasArret	6	INT		pas pour la mesure de l'arrêt
eiPosAtteinte	7	BOOL		position atteinte
ebReset	8	BOOL		reset
eiDistDec	9	INT		recalage distance d'avance
<sorties>				
sbStart	1	BOOL		commande d'avance
siDistM	2	INT		distance d'avance

Figure 9.28: Définition du bloc fonction nMADist sous Unity Pro

9.3.4 Gestion des synchronisations

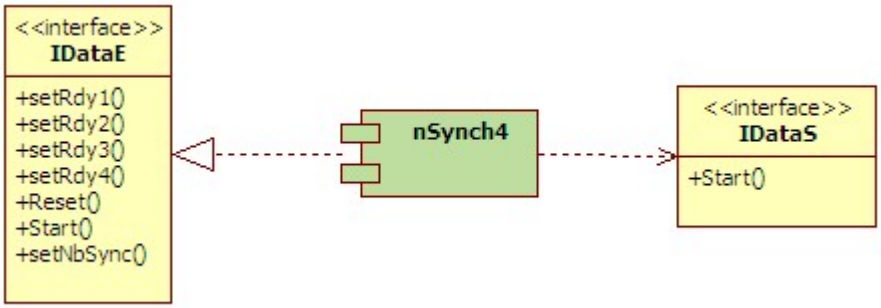


Figure 9.29: Vue logique de l'objet nSynch4

Cet objet permet de déclencher l'appel d'un service « *Start*() » quelconque lorsqu'il aura reçu, via les services « *setRdy*() », des évènements, de 2 à 4 sur cet objet, dont le nombre est défini par « *setNbSync*() ». Par exemple, si on fixe une synchronisation de 4 éléments, il faudra que tous les services *setRdy* soient sollicités avant que *Start* ne soit déclenché.

<entrées>				
ebPret1	1	BOOL		élément 1 prêt
ebPret2	2	BOOL		élément 2 prêt
ebPret3	3	BOOL		élément 3 prêt
ebPret4	4	BOOL		élément 4 prêt
ebReset	7	BOOL		remise à 0
ebStart	8	BOOL		activation du bloc
eiNbSync	9	INT		nombre d'éléments à synchroniser
<sorties>				
sbStart	1	BOOL		top démarrage

Figure 9.30: Définition du bloc fonction nSync4

9.4 **Assemblage des nouvelles briques**

De nouveaux objets ont été spécifiés précédemment en fonction des besoins des personnes réalisant l'analyse descendante du système. Ces derniers ont maintenant à disposition les éléments de contrôle leur permettant de mettre en place la gestion du fonctionnement des cadenceurs.

Nous montrons sur la figure 9.31 un exemple de spécification sous SysML de l'interaction structurelle d'objets d'automatisme via l'utilisation d'un diagramme de structure composite utilisé comme « *ibd* » (*internal block definition* cf 7.2.4). Cette vue permet de préciser les échanges entre les blocs SysML internes à la fonction de cadencement, et notamment de montrer les connexions entre les flux de données de ces sous-éléments.

On voit en particulier comment transite le flux de contrôle. C'est le bloc de synchronisation qui déclenche simultanément les blocs de contrôle Marche/Arrêt. Ces derniers provoquent la mise à jour et l'envoi des ordres d'avances en terme de distances d'avance. Ces distances sont ensuite converties en position pour préparer l'envoi vers des éléments de contrôles fonctionnant avec des positions.

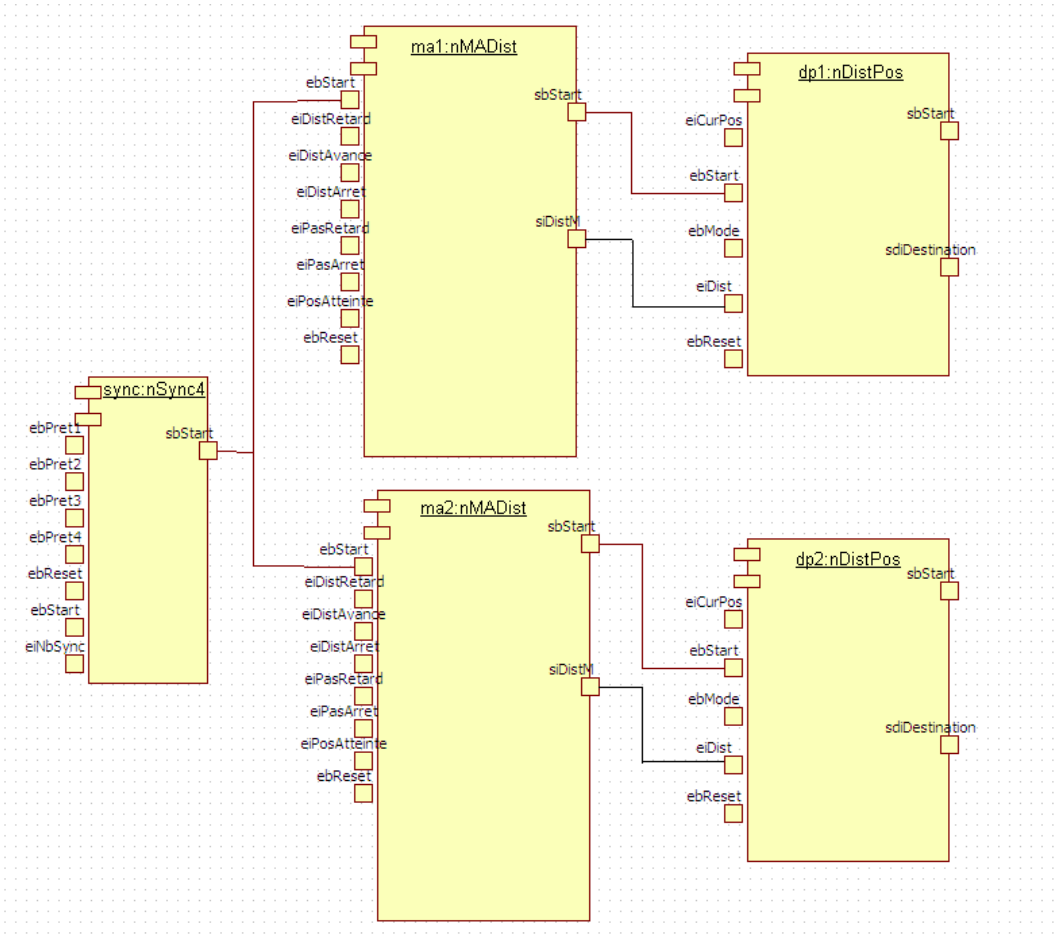


Figure 9.31: Vue Composite du block SysML "Cadencement"

La vue suivante montre l'utilisation de cette composition d'objets dans l'environnement Unity Pro. Les entrées/sorties sont connectées aux éléments externes du programme automate, ce qui donne le diagramme de la figure 9.32.

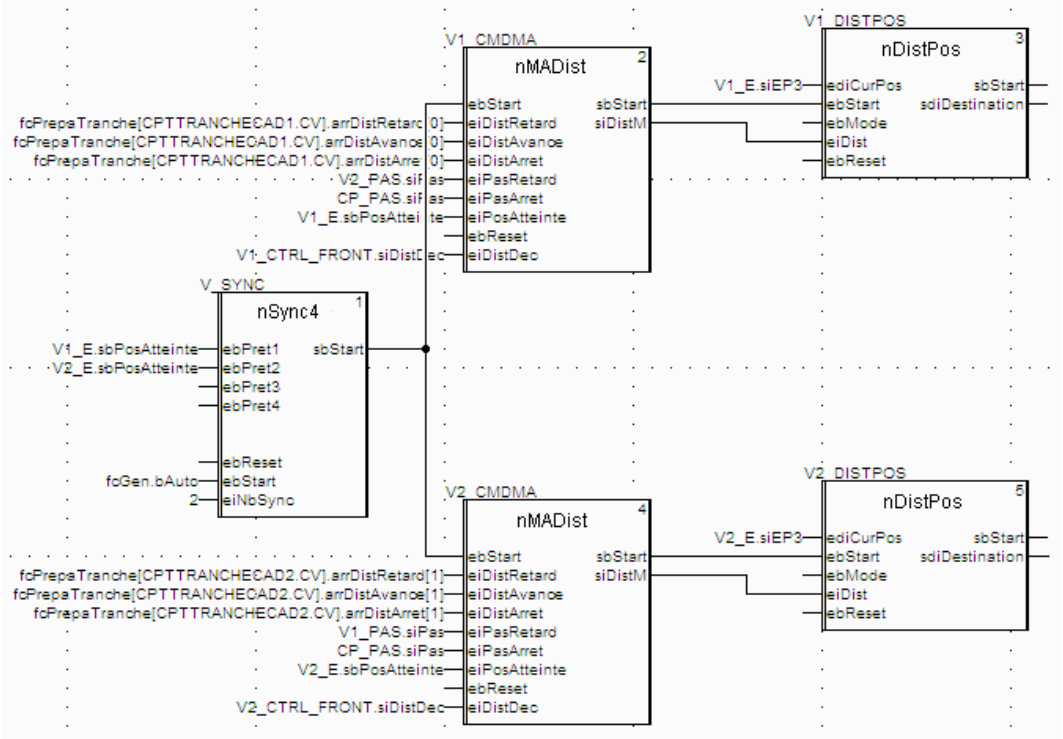


Figure 9.32: Diagramme bloc fonction de la gestion du cadencement sous Unity Pro

9.5 **Remarque sur la description du comportement interne des objets d'automatisme**

Notre objectif est d'obtenir une description, dans le modèle SysML, du comportement dynamique interne des blocs définis précédemment. L'utilisation des diagrammes d'activité SysML permet de représenter les logiques de contrôle internes des différents blocs (cf. 7.3.1.4.2).

Par contre, au niveau du déploiement vers les plates-formes d'exécution, il n'y a pas, pour le moment, de description standardisée des langages textuels tels que le langage **Structuré** ou le langage **LIST**. Dans l'entreprise nous travaillons en interne sur la spécification d'un plug-in SysML nous permettant d'obtenir du code sous une forme proche du langage automate ST, mais pour le moment le code Structuré est intégré manuellement au niveau des fichiers de définition PLCopen comme illustré dans les

copies de fichiers intégrées précédemment.

Pour l'exemple simple de l'objet *nPosDist*, on a un diagramme d'activité sous la forme suivante :

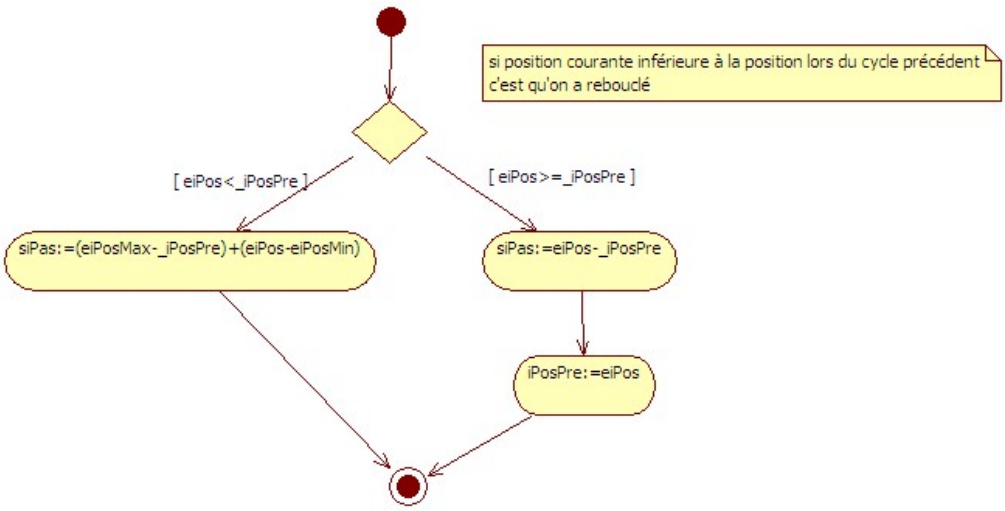


Figure 9.33: Diagramme d'activité de l'objet nPosDist

Ce diagramme est traduit manuellement en langage ST dans le fichier PLCopen puis conservé tel quel jusqu'à la conversion en fichier de définition Unity Pro. Dans la définition du bloc fonction *nPosDist* sous Unity Pro, on retrouve ensuite le code sous cette forme :

```
(*si position courante inférieure à la position lors du cycle précédent
c'est qu'on a rebouclé*)
if eiPos<_iPosPre then
  siPas:=dint_to_int((eiPosMax-_iPosPre)+(eiPos-eiPosMin));
else
  siPas:=dint_to_int(eiPos-_iPosPre);
  _iPosPre:=eiPos;
end_if;
```

Figure 9.34: Code ST interne au bloc fonction nPosDist

9.6 Autres Remarques

Cette méthodologie de spécification et de déploiement est en cours dans l'entreprise et permet de structurer les standards automatisme sous une forme peu habituelle dans le contexte de l'automatisme traditionnel, une structure à base d'objets. La structuration du standard sous cette forme permet de découpler les réalisations d'un environnement automate cible donné, mais également de découper des fonctionnements en identifiant des sous-fonctions potentiellement réutilisables dans d'autres contextes.

Dans le cas de machines particulièrement complexes, une telle approche permet une meilleure compréhension et une meilleure analyse du système, en plus d'un découpage de la complexité pour faciliter la compréhension des fonctionnements par les concepteurs et les automaticiens.

De plus l'approche présentée permet de placer l'analyse logique comme une facette au sein d'une spécification plus globale du système sous UML, ce qui permet d'améliorer la cohérence entre les différents corps de métier participant à la réalisation des machines.

Bibliographie

[*AES 1989a*] European Space Agency, "*HOOD Reference Manual Issue 3.0*", WME/89-353/JB, 1989.

[*AES 1989b*] European Space Agency, "*HOOD User Manual Issue 3.0*", WME/89-353/JB, 1989.

[*AES 1991*] European Space Agency, "*HOOD Reference Manual Issue 3.1*", HRM/91-07/V3.1, 1991.

[*Backus 1958*] J. Backus, "*Preliminary Report, Specifications for the IBM Mathematical FORmula TRANslating System, FORTRAN*", rapport publié par l'IBM Applied Science Division, 1958.

[*Baudouin 1996*] C. Baudouin, G. Hollowell, "*Realizing the Object-Oriented Lifecycle*", ed. Prenctice Hall, 1996.

[*Berard 1993*] E.V. Berard, "*Essays on Objetc-Oriented Software Engineering*", ed Prenctice Hall, 1993.

[*Berard 1998*] E. V. Berard, "*Object-Oriented Design*", The Encyclopedia of Software Engineering, Volume 2, J.J. Marciniak, Editor, John Wiley and Sons, New York, 1998.

[*Bergenti 2000*] F. Bergenti, A. Poggi, "*Exploiting UML in the Design of Multi-Agent Systems*", First International Workshop on Engineering Societies in the Agent World, 2000.

[*Bitsch 2005*] F. Bitsch, P. Göhner, F. Gutbrodt, U. Katzke, B. Vogel-Heuser, "*Specification of Hard Real-Time Industrial Automation Systems with UML-PA*", International Conference on Industrial Informatics, 2005.

[*Björkander 2003*] M. Björkander, C. Kobryn, "*Architecting Systems with UML 2.0*", magasine IEEE Software, p57-61, July 2003.

[*Blair 1991*] G. Blair, J. Gallagher, D. Hutchison, D. Sheperd, "*Object-Oriented Languages, Systems and Applications*", ed. Halsted Pr, April 1991.

[*Boehm 1988*] B. Boehm, "*A spiral model of software development and enhancement*", magasine IEEE Computer, p 61-72, May 1988.

[*Bonfatti 1995*] F. Bonfatti, P.D. Monari, G. Gadda, "*Bridging structural and software design of PLC-based system families*", Proceedings of the 1st International Conference on Engineering of

Complex Computer Systems, 1995.

[*Bonfè 2000*] M. Bonfè, C. Fantuzzi, "*Mechatronic Objects encapsulation in IEC 61131-3 Norm*", Proceedings of the. IEEE International Conference on Control Applications, 2000.

[*Bonfè 2001*] M. Bonfè, C. Fantuzzi, "*Object-oriented approach to PLC software design for a manufacturemachinery using IEC 61131-3 norm languages*", Proceedings of the. IEEE International Conference on Advanced Intelligent Mechatronics, 2001.

[*Bonfe 2005*] M. Bonfe, C. Fantuzzi, C. Secchi, "*Behavioural inheritance in object-oriented models for mechatronics systems*", International Journal of Manufacturing Research - Vol. 1, No.4 pp. 421 – 441, 2006.

[*Booch 1991*] G. Booch, "*Object Oriented Design with Applications*", ed. Benjamin-Cummings, 1991.

[*Borges 1997*] J. Borges," *PC vs. PLC for Machine and Process Control*", Real Time Magazine, n°97-4 p71-72, 1997

[*CEA 2003*] CEA, I-Logix, Uppsala, OFFIS, PSA, MECEL, ICOM, "*UML based methodology for real time embedded systems*", délivrable AIT-WOODDES, 2003.

[*CEA 2006*] D. Servat, "*Guide méthodologique de modélisation par composants pour le système CLIPS*", CEA Service Outils Logiciels, délivrable CLIPS, 2006.

[*Chiron 2005*] F. Chiron, K. Kouiss, "*Distributed control systems : from design to realimplementation with UML 2.0*", Proceedings of the International Conference on Industrial Engineering and Systems Management, 2005.

[*Chiron 2007*] F. Chiron, K. kouiss, "*Design of IEC 61131-3 Function Blocks using SysML*", Proceedings of the Mediterranean Conference on Control & Automation, 2007.

[*Clark 2004*] T. Clark, A. Evans, P. Sammut, J. Willans, "*Applied Metamodelling, A Foundation for Language Driven Development*", sur le site Xactium ou , « http://www.uio.no/studier/emner/matnat/ifi/INF5120/v06/undervisningsmateriale/AppliedMetamodelling.pdf », 2004.

[*Clements 2001*] Clements P., Northrop L., "*Software Product Lines: Practices and Patterns*", ed. Addison-Wisley, 2001.

[*Coad 1990*] P. Coad, E. Yourdon, "*Object Oriented Analysis*", ed.

Yourdon Press, 1990.

[*Coad 1991*] P. Coad, E. Yourdon, "*Object Oriented Design*", ed. Yourdon Press, 1991.

[*Coggins 1990*] J.M. Coggins, "*Designing C++ Class Libraries*", Proceedings of the C++ Conference, 1990.

[*Dahl 1966*] O. J. Dahl, K. Nygaard," *SIMULA, an ALGOL-Based Simulation Language*", magasine Communications of the ACM, n°9 p671-678, 1966.

[*DOMINO 2007*] IRIT, "*Projet DOMINO*", Site web http://neptune.irit.fr/Public/AnglaisV2/Domino/ficheResumeDOMINO.html., 2007.

[*Dori 2002*] D. Dori, "*Why significant UML Change is Unlikely*", magasine Communications of the ACM, n°45 p82-85, 2002.

[*Douglass 1999*] B. Douglass, "*Doing hard time : developping real-time systems with UML, objects, frameworks and patterns*", Object Technology Series, ed. Addison-Wesley, 2000.

[*Douglass 1999b*] B. Douglass, "*ROPES: Rapid Object-Oriented Process for Embedded Systems*", I-Logix White Papers, 1999.

[*Doumeingts 1984*] G.Doumeingts, "*Méthode GRAI : méthode de conception des systèmes en productique*", Thèse de doctorat, Université de Bordeaux 1, 1984.

[*Dubinin 2004*] V. Dubinin, V. Vyatkin, "*UML-FB, A Language for Modeling and Implementation of Industrial-Process Measurement and Control Systems on the Basis of IEC 61499 Standard*", Proceedings of the New Information Technologies And Systems conference, 2004.

[*Espinasse 1994*] B. Espinasse, M. Lai, D. Nanci, "*MERISE+ an extension of MERISE toward object oriented HOOD method*", Publications de la conférence Informatique des Organisations et Systèmes d'Information et de Décision, 1994.

[*Flores 2003*] L. Flores, J. Barata, "*Object oriented software engineering for programmable logical controllers: a successful implementation*", Proceedings of the IEEE Conference on Emerging Technologies and Factory Automation, 2003.

[*Galland 2001*] S. Galland, "*Approche multi-agents pour la conception et la construction d'un environnement de simulation en vue de l'évaluation des performances des ateliers multi-sites*", Thèse de Doctorat, Université Jean Monnet de Saint Etienne, 2001.

[*Gerard 2000*] S. Gerard, "*Modélisation UML exécutable pour les systèmes*

embarqués de l'automobile", Thèse de Doctorat, Université d'Évry, 2000.

[*Goldberg 1983*] A. Goldberg, D. Robson, M.A. Harrison, "*Smalltalk-80, the language and its implementation*", ed. Addison-Wesley, 1983.

[*Gomaa 1984*] H. Gomaa," *A Software Design Method for Real-Time Systems*", Communications of the ACM, n°9 p938-949, 1984.

[*Gomaa 2000*] H. Gomaa, "*Designing concurrent, distributed, and real-time applications with UML*", Object Technology Series. ed. Addison-Wesley, 2000.

[*Gomaa 2000b*] H. Gomaa, "*Designing Real-Time Applications with the COMET/UML Method*", « http://wooddes.intranet.gr/papers/gomaa.pdf », 2000.

[*Guiochet 2003*] J. Guiochet, "*Maîtrise de la sécurité des systèmes de la robotique de service. Approche UML basée sur une analyse du risque système*", Thèse de Doctorat, Université Joseph Fourier de Grenoble, 2003.

[*Harel 1985*] D. Harel, A. Pnueli, "*On the Development of reactive systems. Logic and Models of Concurrent Systems, NATO. Advanced Study Institute on logics and Models for Verification and Specification of Concurrent Systems*", Lecture Notes in Computer Science, ed. Springer Berlin / Heidelberg, p294-308, 1985.

[*Harel 1987*] D. Harel, "*Statecharts, a visual formalism for complex systems*", Science of Computer Programming, ed. Elsevier North-Holland , p231-274, 1987.

[*Hatley 1987*] D.J. Hatley, I.A. Pirbhai, "*Strategies for Real-Time System Specification*", ed. Dorset House, 1987.

[*Heverhagen 03*] T. Heverhagen, R. Tracht, R. Hirschfeld, "*A Profile for Integrating Function Blocks into the Unified Modeling Language*", Proceedings of Specification and Validation of UML Models for Real Time Embedded Systems, SVERTS, 2003.

[*Heverhagen 2001*] T. Heverhagen, R. Tracht, "*Integrating UML-RealTime and IEC 61131-3 with Function Block Adapters*", Proceedings of the Fourth International Symposium on Object-Oriented Real-Time Distributed Computing, p395, 2001.

[*Hussain 2006*] T. Hussain, G. Frey, "*UML-based Development Process for IEC 61499 with Automatic Test-case Generation*", Proceedings of the Conference on Emerging Technologies

and Factory Automation, ETFA, 2006, .

[*Ichbiah 1983*] J. Ichbiah, "*Reference manual for the Ada programming language*", ed. Springer-Verlag, 1983.

[*IEC 60848 1988*] International Electrotechnical Commission, "*Langage de spécification GRAFCET pour diagrammes fonctionnels en séquence*", IEC Norms, 1988.

[*IEC 61131 1993*] IEC, "*Programmable Controllers Standard*", IEC Norms, 1993.

[*IEC 61499 2005*] International Electrotechnical Commission, "*Function Blocks*", IEC Norms, 2005.

[*Jacobson 1992*] I. Jacobson, M. Christerson, P. Jonson, G. Overgaard, "*Object-Oriented Software Engineering: A Use Case Driven Approach*", ed. Addison-Wesley, 1992.

[*Jacobson 1999*] I. Jacobson, G. Booch, J. Rumbaugh, "*The Unified Software Development*", ed. Addison-Wesley, 1999.

[*Jalote 1989*] P. Jalote," *Functional Refinement and Nested Objects for Object-Oriented Design*", magasine Transactions on Software Engineering, n°3 p264-270, 1989.

[*Jimenez 2001*] F. Jimenez, "*Conception sûre des automatismes industriels: modélisation synchrone de langages d'automates programmables de la norme CEI-61131-3*", Thèse de Doctorat, Université de Grenoble, 2001.

[*Katzke 2005*] U. Katzke, B. Vogel-Heuser, "*UML-PA as an engineering Model for distributed process automation*", Proceedings of the International Federation of Automatic Control Conference, IFAC, 2005.

[*Kemeny 1964*] J. G. Kemeny, T. E. Kurtz, "*A manual for BASIC, the elementary algebraic language designed to use with the Dartmouth Time Sharing System*", ed. Dartmouth Publications , 1964.

[*Kernighan 1983*] B. Kernighan, D. Ritchie, "*The C Language*", ed. Prentice Hall, 1983.

[*Kobryn 1999*] C. Kobryn, "*UML 2001: a Standardization Odyssey*", Communications of the ACM, n°42, p29-37, 1999.

[*Kobryn 2001*] C. Kobryn," *Will UML 2.0 Be Agile or Awkward*", Communications of the ACM, n°1, p107-110, 2001.

[*Kouiss 2006*] K. Kouiss, F. Chiron, D. Servat, T. Meurisse, "*A component based development of user interfaces for manufacturing*

systems", Proceedings of the Conference on Integrated Design and Manufacturing in Mechanical Engineering, IDMME, 2006.

[*Kruchten 1999*] P. Kruchten, "*The Rational Unified Process - an introduction*", Object Technology Series. ed. Addison-Wesley, 1999.

[*Li 1991*] X. Li," *Integration of Structured and Object-Oriented Programming*", Journal of Object- Oriented Programming-Focus on Analysis and Design, n° p54-60, 1991.

[*Lukman 1991*] J. T. Lukman, "*Transforming the 2167A Requirements Definition Model Into an Ada-Object-Oriented Design*", Annual National Conference on Ada Technology, 1991.

[*Maffezzoni 1999*] C. Maffezzoni, L.L. Ferrarini, E. Carpanzano," *Object-oriented models for advanced automation engineering - modular modeling in an object oriented database*", magasine Control Engineering Practice, n°8 p957-968, 1999.

[*Maler 1999*] O. Maler, "*On the programming of industrial computers*", PLCopen White Papers, 1999.

[*Marca 1988*] D.A. Marca, C.L. McGowan, "*SADT - Structured Analysis and Design Technique*", ed. McGraw-Hill, 1988.

[*Marcotte 1995*] F. Marcotte, "*Contribution à la modélisation des systèmes de production : extension du modèle GRAI*", Thèse de doctorat, Université de Bordeaux I, 1994.

[*Mertins 1995*] H. Mertins, R. Jochem, "*A tool for Object-oriented modeling and analysis of business processes*", magasine Computers in Industry, n°33, p 345-356, September 1997.

[*Mullin 1989*] M. Mullin, "*Object-Oriented Program Design: With Examples in C++*", ed. Addison-Wesley, 1989.

[*Myers 1975*] G.J. Myers, "*Reliable Software through Composite Design*", ed. Van Nostrand Reinhold Company, 1975.

[*Myers 1976*] G.J. Myers, "*Software Reliability: Principles and Practices*", ed. John Wiley & Sons, 1976.

[*Myers 1978*] G.J. Myers, "*Composite, Structured Design*", ed. Van Nostrand Company, 1978.

[*OCDE 1992*] Organisation de coopération et de développement économiques, "*Manuel d'Oslo*", 1992.

[*Odell 1999*] J. Odell, C. Bock, "*Suggested UML Extensions for Agents*", response to the OMG Analysis and Design Task Force UML

RTF 2.0 Request for Information, 1999.

[*Odell 2000*] J. Odell, H. Van Dyke Parunak, B. Bauer, "*Extending UML for Agents*", Proceedings of the AOIS Worshop at AAAI, 2000.

[*Ohno 1995*] Ohno T., "*Toyota Production System: Beyond Large-Scale Production*", ed. Productivity Press, 1995.

[*OMG 1999*] OMG, "*Analysis and design platforme task force, white paper on the profile mechanism*", OMG Technical Report, 1999.

[*OMG 2006*] OMG, "*OMG SysML Specification*", OMG Adopted Specification, ptc/06-05-04, 2006.

[*OMG CCM 2006*] OMG, "*CORBA Component Model Specification*", OMG Specifications 2006.

[*OMG MDA 2003*] OMG, "*MDA Guide v1.1*", OMG Specifications, 2003.

[*OMG MOF 2003*] OMG, "*Meta Object Facility Specification*", version 1.3, OMG Specifications, 2003.

[*OMG MOF 2006*] , "*CORBA Component Model Specification v4.0*", OMG Specifications, 2006.

[*OMG UML 2007a*] OMG, "*Unified Modeling Language : Infrastructure*", version 2.1.1, OMG Specifications, 2007.

[*OMG UML 2007b*] OMG, "*Unified Modeling Language : Superstructure*",OMG Specifications, version 2.1.1, 2007.

[*OMG XMI 2005*] OMG, "*MOF 2.0/XMI Mapping Specification, v2.1*", OMG Specifications, 2005.

[*Panjaitan 2006*] S. Panjaitan, G. Frey, "*Combination of UML Modeling and the IEC 61499 Function Block Concept for the Development of Distributed Automation Systems*", Proceedings of the Conference on Emerging Technologies and Factory Automation, ETFA, 2006.

[*Parunak 2001*] H. Van Dyke Parunak, J. Odell, "*Representing Social Structures in UML*", Proceedings of the fifth international conference on Autonomous Agents, AOSE, 2001.

[*Pennaneac'h 2001*] F. Pennaneac'h, "*UML : de l'action à la réflexion*", Thèse de Doctorat, Université de Bretagne Sud, 2001.

[*PLCOPEN 2004*] PLCopen Technical Committee 6, "*XML Formats for IEC 61131-3*", PLCopen Technical Paper, 2004.

[*Remaud 2004*] Remaud P., "*De l'école de la régulation française au début du XXe siècle à l'émergence de l'automatique en France*

après la seconde guerre mondiale", Thèse de Doctorat, Faculté d'Histoire de l'Université de Lyon II, 2004.

[*RF 2003*] Ministère de l'Economie des Finances et de l'Industrie, "*L'excellence en conception dans l'industrie*", Publications du ministère, 2003.

[*Ross 1977a*] D.T. Ross," *Structured Analysis (SA): A Language for Communicating Ideas*", magasine Transactions on Software Engineering, n°3 p16-34, 1977.

[*Ross 1977b*] D.T. Ross, E. Kenneth, J. Schoman," *Structured Analysis for Requirements Definition*", magasine Transactions on Software Engineering, n°3 p6-15, 1977.

[*Roussel 1999*] J-M. Roussel, S. Lampérière Couffin, J-J Lesage, "*IEC 60848 et IEC 61131-3 : deux normes complémentaires*", Journée d'études "nouvelles percées dans les langages pour l'automatique", 1999.

[*Rumbaugh 1991*] J. Rumbaugh, M. Blaha,W. Premerlani, F. Eddy, W. Lorensen, "*Object-Oriented Modeling and Design*", ed. Prentice-Hall, 1991.

[*Selic 1994*] B. Selic, G. Gullekson, P. T. Ward, "*Real-time Object-Oriented Modeling*", ed. John Wiley and Sons, 1994.

[*Selic 1998*] B. Selic, J. Rumbaugh, "*Using UML for Modeling Complex Real-Time Systems*", Proceedings of the ACM SIGPLAN Workshop on Languages, Compilers, and Tools for Embedded Systems, ed. Springer-Verlag, 1998.

[*Servat 2005*] D. Servat, F. Chiron, K. Kouiss, T. Meurisse, G. Souchet, "*Modèle d'architecture générique pour la supervision des systèmes de production*", Publications de la conférence sur l'Ingénièrie Dirigée par les Modèles, 2005.

[*SPEEDS 2006*] B. Esler, K. Mason, A. Williams, "*Initiative SPEEDS, conception de systèmes embarqués*", Communiqué de presse, http ://www.humbugpr.com/speeds/SPE001F-Launch-FR.doc., 2006.

[*Stevens 1974*] W.P. Stevens, G.J. Myers, L.L. Constantine, "*Structured Design*", Classics in software engineering, ed. Yourdon, p205-232, 1974.

[*Stroustup 1991*] B. Stroustrup, "*The C++ Programming Language*", ed. Addison-Wesley, 1991.

[*Su 1997*] Z. Su, "*Automatic Analysis of Relay Ladder Logic Programs*", Technical Reports, University of California at

Berkeley, 1997.

[*T.O. Agency 1995*] T. O. Agency, "*A Comparison of Object-Oriented Methodologies*", T.O. Agency Technical Reports, 1995.

[*Tardieu 1985*] H. Tardieu, A. Rochfeld, R. Colleti, "*La méthode MERISE - Tome I : Principes et Outils*", Edition d'Organisation, ed. Eyrolles, 1985.

[*Taylor 1911*] Taylor F.W., "*The principles of Scientific Management*", ed. Courier Dover Publications, 1911.

[*Taylor 1990*] D.K. Taylor, A. Hecht," *Using CASE for Object-Oriented Design with C++*", magasine Computer Language, n°11 p49-57, 1990.

[*Thramboulidis 2004*] K. Thramboulidis, "*Using UML in Control and Automation, A Model Driven Approach*", Proceedings of INDIN' 04, 2004.

[*US Air Force 1993*] US Air Force, "*Integrated Computer Aided Manufacturing Definition Language (IDEF methods)*", Technical Reports, National Institute of Standard and Technology (NIST), 1993.

[*W3C 2006*] World Wide Web Consortium, "*Extensible Markup Language (XML) 1.0 (Fourth Edition)*", W3C Recommendation, 2006.

[*Ward 1985*] P.T. Ward, S.J. Mellor, "*Structured Development for Real-Time Systems*", ed. Prentice-Hall, 1985.

[*Williams 1994*] T. Williams, "*The Purdue Enterprise Reference Architecture*", magasine Computers in Industry, n°24, p 141-158, 1994.

[*Wirth 1971*] N. Wirth, "*The programming language Pascal*", Computing Center, University of Colorado, 1971.

[*Ziadi 2004*] T. Ziadi, "*Manipulation des Lignes de Produits en UML*", Thèse de Doctorat, Université de Rennes, 2004.

Résumé de thèse « Contribution à la flexibilité et à la rapidité de conception des systèmes automatisés avec l'utilisation d'UML »

La dynamique actuelle des marchés entraîne avec elle une complexité croissante des demandes du client et nécessairement des contraintes de production. Les méthodologies traditionnelles de conception de systèmes montrent leurs limites dans des contextes très changeants pour lesquels les spécifications sont amenées à évoluer rapidement, des éléments technologiques particuliers de réalisation étant souvent pris en compte trop tôt dans le travail d'étude, limitant la versatilité des développements. Les entreprises doivent alors capitaliser au maximum les efforts menés dans les phases amont de spécification pour optimiser les temps d'étude.
Notre travail de recherche s'intéresse plus précisément au domaine des systèmes automatisés et se propose de répondre à la problématique précédente en utilisant des techniques issues du monde de l'informatique pour la réalisation des systèmes physiques, comme l'**OOA** (*Approche Orientée Objet*) et la modélisation objet **UML** (*Langage de Modélisation Unifié*) avec la perspective d'une spécialisation tardive et d'une génération automatique selon les cibles technologiques choisies, comme le préconise la logique **IDM** (*Ingénierie Dirigée par les Modèles*).
L'originalité de ce mémoire est de décrire une méthodologie et une organisation de travail pour la conception des systèmes automatisés, en s'appuyant sur le concept d'objet d'automatisme multi-facettes. De plus, nous proposons une utilisation de l'extension **SysML** (*Langage de Modélisation des Systèmes*) pour la représentation d'éléments d'automatisme particuliers, les *blocs fonctions* de la norme **IEC 61131-3**, à travers le stéréotype " *block* ". Enfin nous montrons comment il est possible d'obtenir une première génération de code automate en passant par les spécifications **PLCopen**, définissant un lien entre une syntaxe **XML** (*Langage de balisage eXtensible*), se voulant standard, et les langages de la norme **IEC 61131-3**. Le passage par cette représentation standardisée permet de garder l'indépendance des implémentations vis-à-vis d'un environnement intégré de développement particulier.
Le processus de conception décrit a été appliqué à un cas d'étude industriel réel appartenant au domaine de la palettisation robotisée.