N° D'ORDRE :

UNIVERSITE PARIS XI
UFR SCIENTIFIQUE D'ORSAY

THESE

Présentée

Pour obtenir

Le GRADE de DOCTEUR EN SCIENCES

DE L'UNIVERSITE PARIS XI ORSAY

PAR

Shebli ANVAR

Sujet : METHODOLOGIE DE DEVELOPPEMENT ET DE MODELISATION UML DES SYSTEMES D'ACQUISITION ET DE TRAITEMENT EN TEMPS REEL POUR LES EXPERIENCES DE PHYSIQUE DES HAUTES ENERGIES

Soutenue le 13 septembre 2002 **devant la Commission d'examen**

M Reza ANSARI .....Président
M Jean-Marc GEIB ..... Rapporteur
M François TERRIER .....Directeur
M Sylvain TISSERANT ..... Rapporteur
M Bertrand VALLAGE

Remerciements

Pour avoir été un directeur de thèse actif et patient, je tiens à remercier tout particulièrement François Terrier. Sans ses encouragements et les échanges fructueux que nous avons eus, ce travail n'aurait jamais vu le jour. Je remercie également François Darnieaud, Philippe Rebourgeard et Michel Mur pour m'avoir autorisé à mener à bien cette thèse en usant d'une partie non négligeable de mon temps de travail au CEA/DAPNIA. Pour avoir accepté de donner de leur temps et de leur énergie pour effectuer le nécessaire travail de rapporteur de mémoire de thèse, je remercie tout particulièrement Jean-Marc Geib et Sylvain Tisserant. Mes remerciements les plus chaleureux également à Reza Ansari et Bertrand Vallage pour leur précieuse participation au jury. Pour leurs encouragements, leur intérêt pour ce travail, ainsi que leur aide matérielle et intellectuelle, je remercie François Bugeon, Denis Calvet, Sébastien Gérard et Irakli Mandjavidzé. Un grand merci également à mes parents, Dominique Camerlynck et Manoutcher Anvar, à mes beaux-parents, Claudette Gauvain et Gilbert Martin, pour leur soutien constant et précieux. Enfin, je tiens à exprimer ma gratitude à mon épouse Stéphanie Martin et au Pr. Bahram Elahi sans le soutien et les encouragements desquels, je n'aurais pu entreprendre et mener à bien ce travail.

Table des Matières

I

INTRODUCTION

Cette section introductive est destinée à présenter brièvement : en premier lieu, le contexte des grands instruments de la recherche fondamentale en physique et les problématiques générales rattachées au développement des systèmes électroniques de traitement de l'information liés à ces instruments, en second lieu, l'objectif de la thèse dans ce contexte, et enfin, le plan général du mémoire.

I.A CONTEXTE

Depuis quelques 400 ans, nous savons que les progrès de notre connaissance objective des lois de la nature doit s'appuyer sur l'expérimentation scientifique, seule garante de l'enracinement des sciences dans le réel. Au cours des décennies, l'expérimentation scientifique est passée de son expression la plus élémentaire, à savoir la simple observation des phénomènes par le savant, à la mise en œuvre d'expériences de grande envergure nécessitant la collaboration de plusieurs états.

Les expériences de physique fondamentale touchent par définition aux frontières de la connaissance scientifique. Aussi, au fur et à mesure de l'avancement des connaissances, a-t-il fallu détecter et caractériser des phénomènes de plus en plus rares, subtils et fugitifs, ce qui a imposé l'usage de technologies permettant de traiter massivement un grand nombre de signaux : les expériences d'aujourd'hui sont toutes fondées sur la détection et le traitement électronique de signaux issus de détecteurs physiques, photomultiplicateurs, chambres à dérive, matrices de pixels, calorimètres ou autres capteurs. Il s'est donc développé autour de ces expériences, toute une activité spécifique consacrée à la conception et réalisation de systèmes électroniques de traitement de l'information. Ces systèmes sont composés de multiples sous-systèmes de natures différentes dévolus notamment à la détection, au traitement et à la mise en forme analogiques, à la numérisation, et enfin, à l'acquisition, au traitement et à la mise en forme numériques. Le développement de ces sous-systèmes relève de l'ingénierie en électronique analogique et numérique ainsi qu'en informatique temps-réel et systèmes embarqués. S'ajoute à cela le traitement hors ligne des données pour l'analyse physique et l'exploitation scientifique de l'expérience.

La complexité et le gigantisme croissant des expériences des systèmes de détection permettant de sonder de plus en plus loin les horizons de l'infiniment petit et de l'infiniment grand a imposé le recours à des collaborations internationa-

les, seules capables de rassembler les ressources financières et humaines nécessaires. Ainsi, le modèle de fonctionnement des expériences de physique fondamentale repose sur la notion de « grand instrument » international pouvant accueillir de multiples expériences proposées par des collaborations de laboratoires de recherches de part le monde. Ce grand instrument peut aussi bien être un accélérateur de particules tels le collisionneur du CERN* à Genève [A-1], qu'un télescope optique tels ceux de l'ESO à Paranal [A-2] ou qu'un instrument particulier tel le télescope à neutrino ANTARES [A-3][1]. De nombreux laboratoires de recherche dans le monde sont donc amenés à concevoir et réaliser des sous-systèmes de détection, acquisition et traitement temps réel originaux pour les expériences de physique. Le Service d'Electronique des Détecteurs et d'Informatique (SEDI*) du Département d'Astrophysique, physique des Particules, physique Nucléaire et Instrumentation Associée (DAPNIA*), structure d'accueil de ce travail de thèse, est un tel laboratoire. Ses compétences recouvrent tous les sous-systèmes évoqués ce qui confère au laboratoire une vision verticale de toute la chaîne d'instrumentation des expériences de physique. A ce titre, le SEDI est en particulier confronté aux problématiques liées au développement des sous-systèmes numériques dévolus à *l'acquisition* et au *traitement en temps réel* des flots de données issus de la numérisation des signaux provenant des détecteurs de physique. Pour alléger l'écriture, nous désignons ces sous-systèmes par l'acronyme TDAQ*.

Dans ce travail de thèse, nous nous pencherons donc sur ces problématiques de développement de systèmes TDAQ*. Nous tenterons d'en exprimer les spécificités –notamment en matière de distribution et de parallélisme– et les structures récurrentes afin d'établir les problèmes typiques de spécification, conception, réalisation, maintenance et évolution rencontrés lors du développement de tels systèmes. Notre domaine d'étude touche donc au génie logiciel ou, plus généralement, au génie des systèmes électroniques de traitement de l'information puisque l'essentiel de l'activité d'ingénierie autour des systèmes TDAQ consiste à concevoir et réaliser un sous-système complet matériel et logiciel, en interaction harmonieuse avec le reste de l'instrument. Nous tiendrons ainsi compte de la question du développe-

* *Tous les termes et expressions spécialisés ainsi que les abréviations spécifiques sont définis en section VIII « Termes, acronymes et abréviations ». Ils seront signalés par un astérisque au moins lors de leur première occurrence dans l'ouvrage.*

1. *Les sites internet institutionnels de ces grands instruments sont fournis par les références [A-1] à [A-17]*

ment conjoint matériel et logiciel dans ses problématiques particulières aux systèmes TDAQ, sans toutefois aborder les problèmes de partitionnement par simulation de performance, certes classiques aux activités de codesign, mais situés en dehors du domaine d'intérêt de cette thèse.

I.B OBJECTIFS

L'évolution des systèmes TDAQ* montre une tendance vers un gigantisme et une complexité accrus, appelant ainsi une évolution *ad hoc* des méthodologies et outils de développement, sous peine de remettre en cause la faisabilité des systèmes du futur. Cette extension continue en taille et complexité mène tout naturellement à des problèmes d'intelligibilité, de coût de développement et de maintenance des systèmes, que l'on ne pourra gérer qu'en faisant évoluer les processus de développements de manière à produire des composants logiciels plus modulaires et plus réutilisables, ainsi que des architectures de systèmes plus intelligibles et plus rationnelles. Nous nous proposons donc de mener une analyse de domaine qui, d'une part, fasse ressortir les concepts clés et méthodes de développement classiquement mis en œuvre pour la réalisation de systèmes TDAQ*, et d'autre part, mette en lumière les confusions sémantiques et les erreurs méthodologiques qui nuisent à la modularité et l'intelligibilité des systèmes développés. En pratique, il s'agira de dégager des réponses aux problèmes liés à l'articulation entre *la spécification de principe* des systèmes TDAQ* et leur *conception/réalisation* sur une plate-forme matérielle et logicielle concrète.

L'axe principal de notre travail sera la définition d'une *méthodologie de développement des systèmes TDAQ* qui réponde aux problématiques de développements particulières à ces systèmes. Notre proposition se présentera avant tout comme un *processus de développement* dont l'usage régulier permettra à une équipe de développement ou un service d'ingénierie de mettre en place progressivement un outil informatique matérialisant ce processus sous la forme d'un canevas*, c'est-à-dire un atelier de développement comprenant aussi bien des composants réutilisables que des éléments d'architecture génériques adaptés aux applications TDAQ. L'objectif de ce mémoire est donc la spécification d'un processus de développement de systèmes TDAQ, suffisamment détaillée et formalisée pour pouvoir donner lieu à la réalisation d'un *« méta-canevas »* ou *« canevas méthodologique »* dont l'usage par une équipe de développement de système TDAQ doit permettre

la production d'un canevas TDAQ « maison ».

Nous tenterons également de dégager dans quelle mesure cette solution pourrait être étendue au développement de systèmes temps réel en général, tout en en soulignant les différences par rapport aux méthodologies de développement généralistes de l'industrie informatique.

I.C PLAN DU MÉMOIRE

Nous présenterons en section II les caractéristiques et les évolutions des systèmes TDAQ pour les expériences de physique et nous tenterons de justifier la nécessité d'une réflexion approfondie sur les méthodes de développement dans ce domaine. Puis, nous nous intéresserons plus particulièrement en section III aux technologies susceptibles d'apporter une réponse aux problématiques évoquées. Nous tenterons d'exposer un état de l'art sur les méthodologies de développement, et les techniques de réutilisation logicielle telles les motifs récurrents* et les canevas*, avec une orientation en faveur des domaines du temps-réel et de l'embarqué. Nous exposerons ensuite en section IV les développements conceptuels spécifiques accomplis par ce travail de thèse dans le cadre de cette réflexion. Y seront exposés notamment le modèle de développement correspondant à l'approche proposée, certaines structures générales qui se retrouvent de manière récurrente dans les systèmes TDAQ ainsi qu'une discussion sur ce qui, pour ces systèmes, est à retenir dans les modèles de développement généralistes proposés par l'industrie logicielle. Enfin, nous terminerons notre propos en section V en envisageant les travaux futurs associés à cette thèse. Nous évoquerons notamment le projet de R&D MORDICUS* dont le but et d'aboutir à une mise en œuvre concrète des concepts et idées élaborés dans ce travail ainsi que les développements récents de l'OMG* sur les « architectures orientée modèles », particulièrement en résonance avec les concepts développés dans cette thèse.

II

UN CANEVAS TDAQ : POURQUOI ET COMMENT ?

Un système électronique de traitement de l'information est avant tout une application fabriquée dans le but de rendre à des utilisateurs un service relevant du traitement de l'information. Généralement, il n'y a que cet aspect de l'application – qui est aussi sa raison d'être – qui soit présent dans la psychologie de l'utilisateur. Celui-ci peut en apprécier les performances, la facilité d'utilisation, l'ergonomie, comme il peut en déplorer les insuffisances, les bogues et autres dysfonctionnements. L'ensemble des appréciations portées par les utilisateurs sur l'application est une mesure de ce que l'on peut appeler sa « qualité externe ». En effet, l'utilisateur n'a en général pas à connaître et ne connaît pas la réflexion, les idées, les efforts et la coordination qui ont été nécessaires au bon déroulement du processus de développement et, *in fine*, à la production de l'application.

Pourtant, il est clair que cette qualité externe de l'application, son adéquation à l'attente et aux besoins de l'utilisateur, sont largement tributaires du cheminement poursuivi par les développeurs pour aboutir au produit fini. L'abondance de bogues, par exemple, dépendra directement de la méthode avec laquelle les développeurs auront subdivisé leur application en éléments modulaires simples ainsi que de l'effort qu'ils auront consenti à élaborer des scénarios de test les plus exhaustifs possibles. Pourquoi une grande part de la littérature sur le génie logiciel est-elle si soucieuse d'imaginer des méthodes, des outils, des concepts, des langages, des architectures et des cadres conceptuels dont les seuls buts sont la « modularité » et la « maintenabilité » ? Parce que ces caractéristiques appartiennent à ce qu'on pourrait appeler la « qualité interne » de l'application, c'est-à-dire l'ensemble de ses caractéristiques qui, bien qu'invisibles à l'utilisateur, vont déterminer plus ou moins directement sa qualité externe.

L'un des arguments majeurs avancés en faveur de la qualité interne des logiciels est l'argument économique : on souligne souvent dans la littérature du génie logiciel (voir par exemple [C-7] [C-8]) que « qualité logicielle » est, à plus ou moins long terme, synonyme de « réduction de coûts » et donc générateur de profit. De même, l'un des arguments favoris des vendeurs d'AGL* et de composants intégrés est la réduction du « *time-to-market* » permettant aux entreprises une plus grande réactivité face à un marché en évolution rapide et donc une amélioration de leurs ventes grâce à une meilleure adéquation aux besoins de leurs clients. Bien que difficiles à mesurer, il semble que ces arguments économiques soient bien légitimes

puisque durant ces dix dernières années, l'industrie logicielle a mis en pratique avec succès les techniques d'amélioration de la qualité interne des logiciels.

La mesure du succès d'une activité de recherche fondamentale n'est bien entendu pas pécuniaire puisque le produit final de cette recherche est la connaissance scientifique générée et non un profit. Elle est néanmoins concernée par la qualité interne des systèmes de traitement de l'information qu'elle utilise pour son activité, car elle doit se soucier d'une part, de réduire les coûts qu'elle génère afin d'améliorer sa productivité scientifique, et d'autre part, de minimiser la perte de qualité de ses résultats scientifiques due à l'appareillage. Nous verrons au II.A.4, par exemple, que le comportement d'un système TDAQ peut avoir une influence déterminante sur la qualité du résultat scientifique et que l'analyse scientifique des données d'une expérience nécessite de connaître en détail ce comportement. Il est par conséquent crucial que l'architecture du système TDAQ et son code source soient suffisamment intelligibles pour que tout physicien qui se penche sur l'analyse des données en ait une vision pertinente.

Dans cette section, nous tentons de justifier en quoi l'élaboration d'un canevas spécifique au développement des systèmes TDAQ est, sinon nécessaire, du moins souhaitable. Nous verrons d'abord (II.A) la définition et les caractéristiques d'un système TDAQ, puis les caractéristiques de leur cycle de développement (II.B) et enfin (II.C), quelles évolutions connaissent ces systèmes et en quoi leur l'ingénierie de développement en est affectée. S'appuyant sur ces sous-sections, la dernière sous-section (II.D) énoncera brièvement en quoi une rationalisation accrue des processus de développement TDAQ est nécessaire et comment nous comptons nous y prendre pour apporter une réponse pertinente à cette question.

II.A CARACTÉRISTIQUES DES SYSTÈMES TDAQ

Les systèmes TDAQ rencontrés dans les multiples expériences de physique des hautes énergie et d'astroparticules sont d'une très grande variété. La palette des questions scientifiques abordées étant extrêmement vaste, les caractéristiques des détecteurs et de leurs systèmes d'acquisition et de traitement en ligne sont à l'avenant. Il est cependant possible de dégager des structures et des caractéristiques récurrentes, communes aux systèmes TDAQ en général, sans préjuger de leur implémentation* matérielle ou logicielle particulière.

II.A.1 Systèmes d'acquisition « déclenchés »

Le but des expériences de physique fondamentale est de détecter ou mesurer des phénomènes situés aux frontières de la connaissance scientifique, avec comme conséquence pratique que les signaux recherchés sont – par définition – difficiles à produire et à détecter. Le « rapport signal / bruit » au sens large, c'est-à-dire la part des signaux correspondant effectivement au phénomène recherché par rapport aux signaux correspondant aux phénomènes connus, est donc, dans ces expériences, extrêmement faible. L'expérience ANTARES par exemple [B-2], s'attend à détecter une dizaine d'événements intéressants par an alors que le taux primaire d'événements dans le détecteur est de 60 kHz, ce qui donne un rapport signal / bruit final de 10^{-12} ! Par ailleurs, la physique moderne étant fondamentalement de nature probabiliste, les mesures effectuées sont de type statistique et leur précision dépend donc directement de l'abondance du signal utile. Il s'agit donc de collecter la plus grande quantité possible de signaux utiles à travers des détecteurs qui fournissent une grande majorité de signaux inutiles. La résolution de ce paradoxe est apportée par les « systèmes de déclenchement* » ou « *triggers* » (dans le jargon de la physique des hautes énergies) qui sont donc le complément indispensable des systèmes d'acquisition des grands détecteurs de physique.

Le terme de « déclenchement » signifie que toute acquisition de données doit être déclenchée par un signal ; ce signal, produit par un calcul en temps réel sur les données sortant du détecteur, indique quelles données, parmi toutes celles produites, sont susceptibles d'intéresser l'expérience. Les données pour lesquelles l'acquisition n'a pas été déclenchée sont donc irrémédiablement perdues. La figure II-1 représente une telle architecture.

Les données brutes en entrée du système d'acquisition résultent de la numérisation et de l'étiquetage temporel des signaux analogiques en provenance des différents sous-détecteurs (chambres à dérive, calorimètres, etc…). Ces données représentent l'état des détecteurs au cours d'un temps de référence fourni par une horloge répartie sur tous les sous-détecteurs (Cf. II.A.2). Elles sont stockées dans des mémoires tampon en attendant que leur sort soit décidé par le sous-système de déclenchement associé.

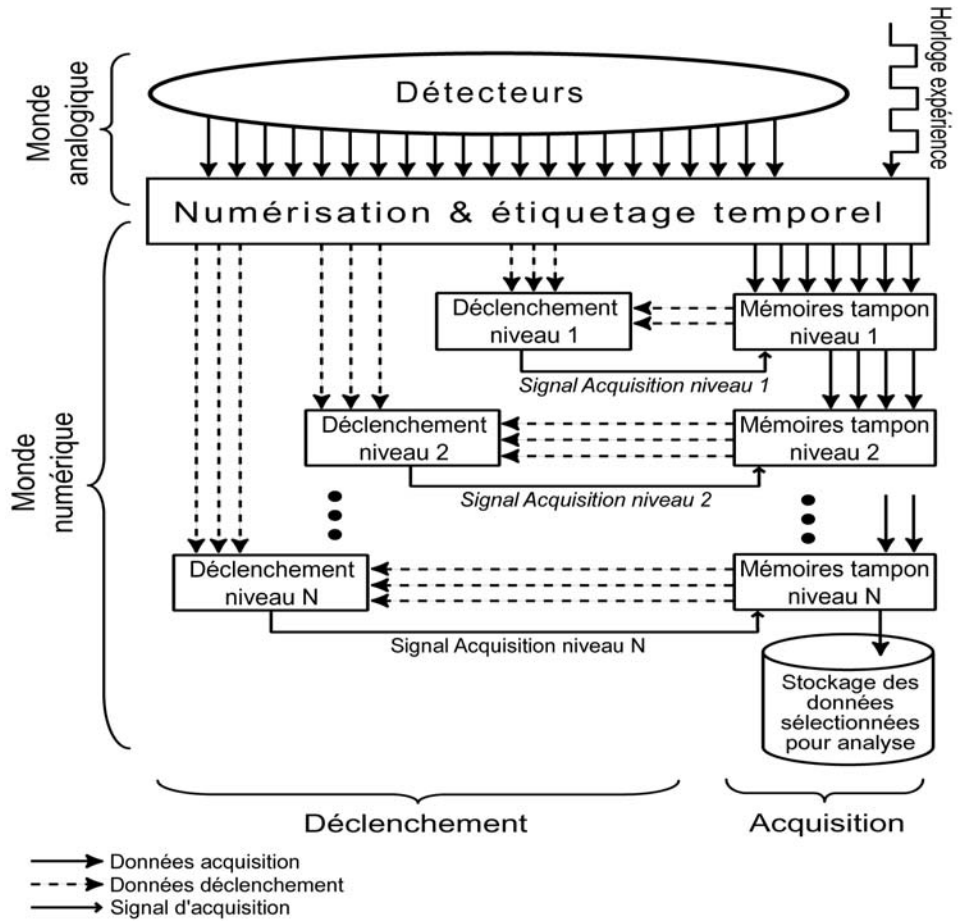


Fig. II-1 : Architecture générale d'un système TDAQ

L'essentiel du flot de données dans un système TDAQ est *monodirectionnel*, depuis les détecteurs jusqu'au stockage sur mémoire de masse. Un système d'acquisition déclenché comprend en général une cascade de plusieurs niveaux de déclenchement, chaque niveau ayant ses propres caractéristiques en termes de taux d'événements, latence, taille mémoire, etc. A la sortie du premier niveau, les données sont regroupées en événements physiques. En d'autres termes, le déclenchement de niveau 1 extrait des données brutes (état des détecteurs) celles qu'il considère comme associées à un événement physique intéressant. Les niveaux de déclenchement suivants affinent l'analyse afin de réduire au maximum le flux de données en rejetant en ligne les événements inintéressants. Cette réduction du flot de données est représentée sur la figure II-1 par la diminution du nombre de flèches pleines en sortie des mémoires tampon.

Les données en entrée des algorithmes de sélection du système de déclenchement peuvent provenir de la chaîne d'acquisition principale aussi bien que d'une

éventuelle chaîne d'acquisition propre au système de déclenchement (les flèches en pointillés sur la figure II-1 représentent les différentes sources possibles).

II.A.2 L'horloge répartie

Toutes les expériences de physique recourent à une référence temporelle précise et unique pour l'ensemble de leurs sous-détecteurs. En effet, l'existence d'une telle horloge est indispensable à la reconstitution des événements de physique à partir des données de détection. La reconstruction de trajectoires d'une particule, par exemple, serait impossible sans la connaissance précise des instants où la particule a produit tel ou tel signal dans tel sous-détecteur. De plus, l'identification et l'étiquetage des événements physiques est fondée sur le temps. En effet, le seul moyen de savoir si différentes données provenant de détecteurs distants appartiennent au même événement physique est d'étiqueter ces données à l'aide d'une même horloge en tenant compte avec précision des délais temporels induits par la répartition géographique des détecteurs. La fréquence de cette horloge doit être d'autant plus grande que le taux d'événements physiques est important afin de réduire la probabilité d'événements multiples associés à une même étiquette temporelle. Sa précision influe directement sur la qualité et la précision des mesures physiques effectuée par le détecteur (résolution spatiale, etc.).

La mesure temporelle est également nécessaire au contrôle des temps moyens d'exécution et au respect des latences de déclenchement (Cf. VI.A). Mais la précision temporelle exigée par cette mesure est en général bien moins élevée que pour la mesure physique : une horloge autre que l'horloge de l'expérience peut donc l'effectuer, même si le plus souvent, elle doit être régulièrement resynchronisée avec celle-ci. Bien entendu, cette mesure n'aura pas lieu d'être si l'on peut garantir que le sous-système aura un comportement suffisamment déterministe, et ce, quelle que soit la structure des événements physiques (plus ou moins complexes) qu'il aura à traiter.

Ainsi, dans l'expérience de physique des particules NA48, tous les signaux en provenances de tous les sous-détecteurs sont étiquetées par rapport à une horloge répartie de 40 MHz [B-4]. Ou encore, l'expérience d'astroparticules ANTARES [B-2] est orchestrée par une horloge globale de 20 MHz dont la précision devra être de l'ordre de 500 ps. Les « événements » physiques y sont définis comme les données produites par l'ensemble du détecteur dans une tranche temporelle.

*II.A.3 L'assemblage d'événement**

Nous avons vu au II.A.2 qu'en amont d'un système TDAQ, des éléments de détection produisent un signal qui est numérisé et dont les données résultantes sont injectées dans l'acquisition (fig. II-1). Or un détecteur de physique est constitué de nombreux canaux de détection géographiquement répartis (Cf. plus loin le II.C.1). Par conséquent, la traversée du détecteur par un événement physique (par exemple le passage d'une gerbe de photons de haute énergie dans un calorimètre) se traduit par la production quasi simultanée de données réparties sur les modules de détection de l'expérience. Ainsi, généralement, en amont de la chaîne d'acquisition, les données appartenant à un même événement physique sont éparpillées sur de multiples nœuds de détection. Il s'agira donc, entre autres, pour le système TDAQ, de *regrouper* les données appartenant à un même événement afin d'une part, de pouvoir procéder à des traitements en ligne faisant intervenir des algorithme physiques (tel des reconstructions sommaires de trajectoires), et d'autre part, d'organiser le stockage des données selon une logique centrée sur les événements physiques. En effet, l'analyse scientifique des données de l'expérience s'appuie sur l'étude statistique des *événements* physiques. Cette opération de regroupement est désignée par l'expression anglaise « *event building* »* que nous avons traduite par « assemblage d'événement »*.

La figure II-2 représente le principe de l'assemblage d'événement : les données

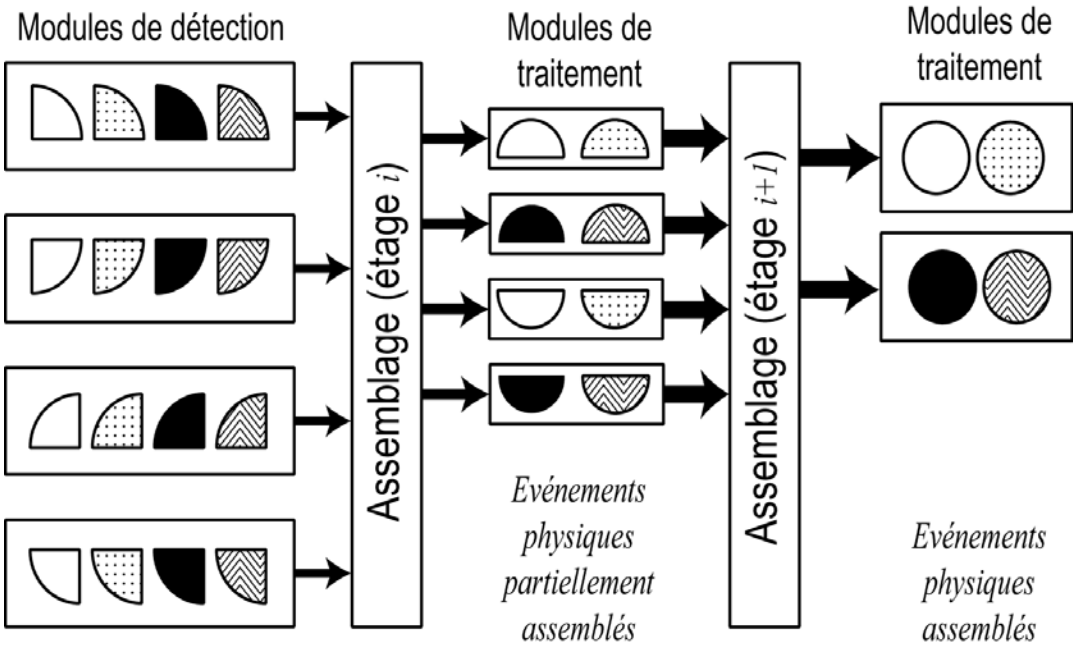


Fig. II-2 : Principe de l'assemblage d'événement

complètes de chaque événement sont représentées par un disque ; en amont de la chaîne d'acquisition, les détecteurs produisent une partie du disque ; en aval, le processus d'assemblage a regroupé ces parties éparses en paquets de données correspondant aux événements complets. Notons également que l'opération d'assemblage peut se faire en plusieurs étapes, chaque étape consistant à procéder à un regroupement partiel de l'événement. De plus, entre chacune de ces étapes, il peut y avoir un étage de traitement avant de poursuivre l'assemblage. La détermination de ces paramètres dépend bien entendu de l'application et des algorithmes de traitement mis en œuvre.

II.A.4 Connaissance du comportement d'un système TDAQ

Les expériences de physique fondamentale visent à effectuer une mesure de type statistique. C'est ce type de mesure qui permet de mettre en évidence une grandeur fondamentale de la nature, une nouvelle particule, des corps célestes quasi invisibles ou autres phénomènes situés à la marge de nos connaissances scientifiques. Or, tout détecteur est par nature imparfait et présente des biais statistiques et des erreurs par rapport à la réalité mesurée. En particulier, les sous-systèmes de déclenchement et – dans une moindre mesure – d'acquisition introduisent toujours de tels biais en raison de la sélection qu'ils opèrent sur les données en fonction d'un traitement algorithmique plus ou moins important. Aussi est-il fondamental pour la validité de l'expérience de connaître ces biais avec une précision compatible avec la précision globale recherchée. Sans cette connaissance, toute mesure statistique fine effectuée par le détecteur n'aurait aucun sens.

II.A.4.1 Nécessité d'une simulation fonctionnelle

Pour connaître le comportement statistique d'un système TDAQ, il est presque toujours fait appel à des programmes de Monte-Carlo dans lesquels on met en œuvre un équivalent fonctionnel des traitements effectués par le système. Connaissant la géométrie du détecteur, les matériaux employés ainsi que la connaissance que nous avons des interactions particules-matière, les Monte-Carlos de la physique fondamentale permettent de générer en grand nombre des données simulant des événements physiques précis : en effectuant sur ces données les mêmes traitements que le système réel effectuerait sur des données réelles, on peut évaluer précisément les caractéristiques statistiques du système TDAQ. Il est donc fondamental de disposer d'une simulation fonctionnellement fidèle du système.

Nous verrons en section IV que cela pose un problème de cohérence entre la simulation Monte-Carlo et la version la plus à jour du système TDAQ dans le processus de développement.

II.A.4.2 Nécessité de mesurer le comportement des systèmes

Un autre type de biais statistique possible est lié aux fautes temporelles du système TDAQ. En effet, si certaines conditions de fonctionnement (par exemple un structure particulière dans la distribution statistique des événements physiques) ont tendance à provoquer des fautes temporelles et donc des pertes de données, alors, si ces conditions de fonctionnement sont corrélées à la nature physique des événements qui les causent, la mesure physique en sera affectée. On peut par exemple imaginer que parmi tous les types d'événements physiques qui intéressent l'expérience, il existe un type dont la complexité particulière provoque plus souvent des fautes temporelles dans le système et est donc plus souvent perdu par l'acquisition que les autres : si la mesure physique fait intervenir un rapport d'abondance entre ce type d'événement et les autres, alors le résultat sera faussé à moins que, justement, l'on en tienne compte dans l'analyse scientifique des données. Il est donc souvent nécessaire d'acquérir une compréhension approfondie du comportement temps réel du système pour résoudre ce type de problème. Cette compréhension peut s'obtenir, dans une certaine mesure, par simulation, mais celle-ci sera surtout destinée à dimensionner les systèmes. En raison de leur trop grande complexité, ceux-ci devront surtout faire l'objet des mesures de comportement directes. Nous verrons ainsi que la mesure du comportement du système occupe une place à part entière dans le processus de développement proposé (Cf. IV.B.6).

II.A.5 Systèmes matériels et logiciels

Les spécificités du domaine des systèmes TDAQ demandent souvent un développement conjoint matériel et logiciel. Principalement pour des questions de réduction de coûts, la tendance générale dans le domaine des grands détecteurs (comme dans beaucoup d'autres) est à utiliser autant que possible des composants COTS*, et à reporter au maximum les développements sur le logiciel afin de minimiser les développements électroniques. La R&D sur les sous-systèmes de l'expérience ATLAS, par exemple, a beaucoup consisté à évaluer la faisabilité des différents sous-systèmes à l'aide de logiciel développé sur des composants

COTS [C-4]. Un autre exemple est le système d'acquisition en mer de l'expérience ANTARES où la conception électronique a été optimisée pour favoriser l'usage de normes répandues et de standards de l'industrie [C-5]. Bien entendu, plus le système à développer sera proche des détecteurs, plus la part matérielle sera importante et spécialisée : dans un système déclenché (Cf. fig. II-1), la conception du sous-système de niveau n aura tendance à comporter plus d'électronique spécialisée que le sous-système de niveau $n+1$.

Retenons également que la plupart du temps, un développement électronique exigera un développement logiciel associé : pilotes logiciels, logiciel de contrôle du module matériel, logiciel de banc de test, etc. En d'autres termes, si un développement purement logiciel est possible (bien que rare pour les systèmes TDAQ), un développement matériel ne peut s'affranchir de la réalisation d'un minimum de logiciel associé.

Enfin, il est à noter qu'aujourd'hui, les réalisations matérielles au sein des systèmes TDAQ sont implémentées le plus souvent (sinon exclusivement) sur des ASICS* ou des FPGA*. La spécification haut niveau de ces implémentations est le plus souvent effectuée aujourd'hui en langage VHDL*, standard de l'industrie, ce qui permet une certaine indépendance de la conception électronique par rapports aux formats propriétaires des différents vendeurs de logiciels de CAO. Aussi est-il plus concevable aujourd'hui d'imaginer une certaine articulation entre conception matérielle et conception logicielle d'un système TDAQ : l'usage de tout environnement et/ou toute méthodologie de développement de la part logicielle d'un système peut, dans une certaine mesure, être étendu à la conception de sa part matérielle à travers le langage VHDL puisque celui-ci s'apparente à un langage informatique.

Au vu de ces éléments, il nous paraît impensable de dissocier les développements électroniques et informatiques d'un même sous-système TDAQ, aussi notre processus de développement de tels systèmes doit-t-il intégrer, au maximum des possibilités techniques, ces deux versants de la conception et de la réalisation des systèmes. Nous verrons ainsi au IV.B comment ceux-ci s'articulent au sein du processus de développement proposé.

II.A.6 Systèmes hétérogènes

Les sous-systèmes mis en œuvre au sein d'une même expérience de physique sont en général très hétérogènes. Les exemples exposés en annexe (Cf. VI.C) mettent en lumière les deux facteurs principaux qui nous semblent être à l'origine de l'hétérogénéité des sous-systèmes TDAQ au sein d'un même projet : le premier consiste en les différences de finalité entre les sous-systèmes, ce qui donne lieu à une « hétérogénéité intrinsèque » ; le second tient essentiellement à la multiplicité des intervenants au sein d'une collaboration scientifique de grande envergure, ce qui est à la base d'une « hétérogénéité circonstancielle ».

II.A.6.1 Hétérogénéité intrinsèque

Comme indiqué dans la figure II-1, les systèmes TDAQ sont composés de plusieurs niveau, chaque niveau comprenant lui-même un sous-système de déclenchement et un sous-système d'acquisition. Au sein d'un même détecteur, les contraintes de débit, de latence et, plus généralement, les contraintes de performance sont *a priori* très différentes pour chacun des sous-systèmes. D'autre part le type de traitement effectué par chacun est également très variable puisque, par exemple, la fonction d'un sous-système d'acquisition est essentiellement la réception, le formatage[1] et la mise en paquet des données, la transmission de ces paquets à travers un réseau et la gestion des flux, alors que celle d'un sous-système de déclenchement est avant tout l'exécution d'algorithmes complexes sur tout ou partie des données aboutissant, pour chaque événement, à un résultat peu volumineux (de l'ordre de quelques octets).

II.A.6.2 Hétérogénéité circonstancielle

Ainsi que souligné dans la section introductive (I), même les détecteurs les plus modestes des expériences de physique fondamentale sont aujourd'hui trop complexes et trop coûteux pour pouvoir être réalisés par un seul laboratoire. Aussi, leur réalisation repose-t-elle sur des collaborations de plusieurs instituts appartenant en général à des pays différents. On désigne ainsi par l'expression « la collaboration* », la communauté des laboratoires et instituts participant à une expérience. La responsabilité de chaque sous-système incombe donc à un laboratoire particulier, possédant ses propres habitudes et sa propre culture technologi-

1. *Le formatage peut comprendre l'exécution d'algorithmes de compression.*

que et l'expérience prouve qu'il est très difficile à plusieurs laboratoires très différents d'adopter le même environnement de développement. En conséquence, les sous-systèmes sont souvent bâtis sur des plate-forme hétérogènes qui compliquent notablement les développements logiciels. Même si ces hétérogénéités circonstancielles pourraient être réduites en principe, les équipes de développement doivent se préparer à les accepter en adaptant leur méthodologie à cette éventualité.

II.A.6.3 Conséquences sur le cadre de développement

Les systèmes TDAQ sont donc immanquablement hétérogènes du fait des deux classes de facteurs sus-cités. Le processus de développement proposé dans ce travail devra donc intégrer à la base la possibilité d'implémentation des systèmes développés sur des plate-forme hétérogènes. Nous verrons, plus particulièrement au IV.B.7, comment la méthodologie proposée traite cette question grâce à la notion de spécification de déploiement.

II.A.7 Des projets longs souvent mis à jour

Les expériences de physique fondamentale sont des projets lourds dont la complexité et l'ambition imposent qu'elles se déroulent sur plusieurs années. Il n'est pas rare de voir un projet s'étaler sur une dizaine d'années, voire plus, depuis la proposition d'expérience jusqu'au démantèlement du détecteur. En effet, même si au départ, le détecteur est conçu pour une certaine classe d'expériences avec des performance données, il arrive presque toujours que de nouvelles idées soient émises au cours de la vie du détecteur pour pouvoir effectuer d'autres expériences en utilisant le même appareillage et moyennant quelques mises à jour et améliorations dans quelques sous-systèmes. Ces prolongations de la vie du détecteur sont en fait une possibilité de faire de la science à moindre coût en complétant l'investissement de départ par le seul coût des améliorations. C'est ainsi que les expériences autour du collisionneur LEP* du CERN* ont eu une durée de vie bien plus grande que celle prévue au départ.

Outre les améliorations dues à de nouvelles idées d'extension de l'expérience, les détecteurs de la physique peuvent également nécessiter des mises à jour et des améliorations pour faire face à des imprévus. L'exemple de NA48 cité en annexe (Cf. VI.D) illustre ce cas de figure.

Les détecteurs de la physique sont donc en général appelés à vivre longtemps et

à connaître des améliorations matérielles et logicielles. Aussi, toute méthodologie de développement des systèmes TDAQ doit-elle assurer une structuration et un processus de développement et de maintenance propices à de telles modifications au cours de la vie du détecteur.

II.B Cycle de développement d'un système TDAQ

Nous avons vu au II.A.7 que les projets de développement de systèmes TDAQ sont des projets longs. Ils sont bien entendu constitués de phases correspondant à des activités différentes. Le déroulement de ces projets n'est aujourd'hui associé à aucun processus formalisé et suit plutôt des schémas heuristiques nés de l'expérience des ingénieurs et physiciens. Nous allons tenter d'en donner une description fidèle tout en en soulignant les points faibles.

II.B.1 Etapes du développement

Une synthèse graphique de la description qui suit est représentée sur la figure II-3, page II-15. Une équipe en charge d'un sous-système[2] va commencer par étudier avec les physiciens concernés les contraintes de flots de données, de puissance de calcul, etc., en un mot les besoins en performances du système final. Deux activités parallèles à l'ingénierie proprement dite mais néanmoins en interaction étroite avec elle (Cf. fig. II-3) sont les campagnes de mesures physiques et les simulations par Monte-Carlo. Ces dernières revêtent une importance particulière car elles sont à la base des estimations des besoins en performances du système TDAQ. Citons, à titre d'exemples, les mesures d'efficacité de détection sur des chambres à dérive prototypes dans l'expérience NA48, ou encore la mesure du niveau de bruit (dû à la radioactivité naturelle et à la bioluminescence) dans l'eau de mer pour l'expérience ANTARES [B-25]. Les résultats de ces campagnes doivent le plus souvent être associés à des simulations par Monte-Carlo des interactions s'effectuant dans les détecteurs afin de produire une vision suffisamment réaliste de la forme et de l'abondance des signaux que le système aura à traiter. Les estimations qu'elles permettent déterminent en grande partie l'élaboration de la topologie et de l'archi-

2. *Nous ne nous étendrons pas sur la partie antérieure du processus de développement qui va justement permettre à la collaboration* de découper l'instrument en sous-systèmes et en assigner la responsabilité du développement aux différents laboratoires participants.*

tecture matérielle du système TDAQ ainsi que des choix technologiques.

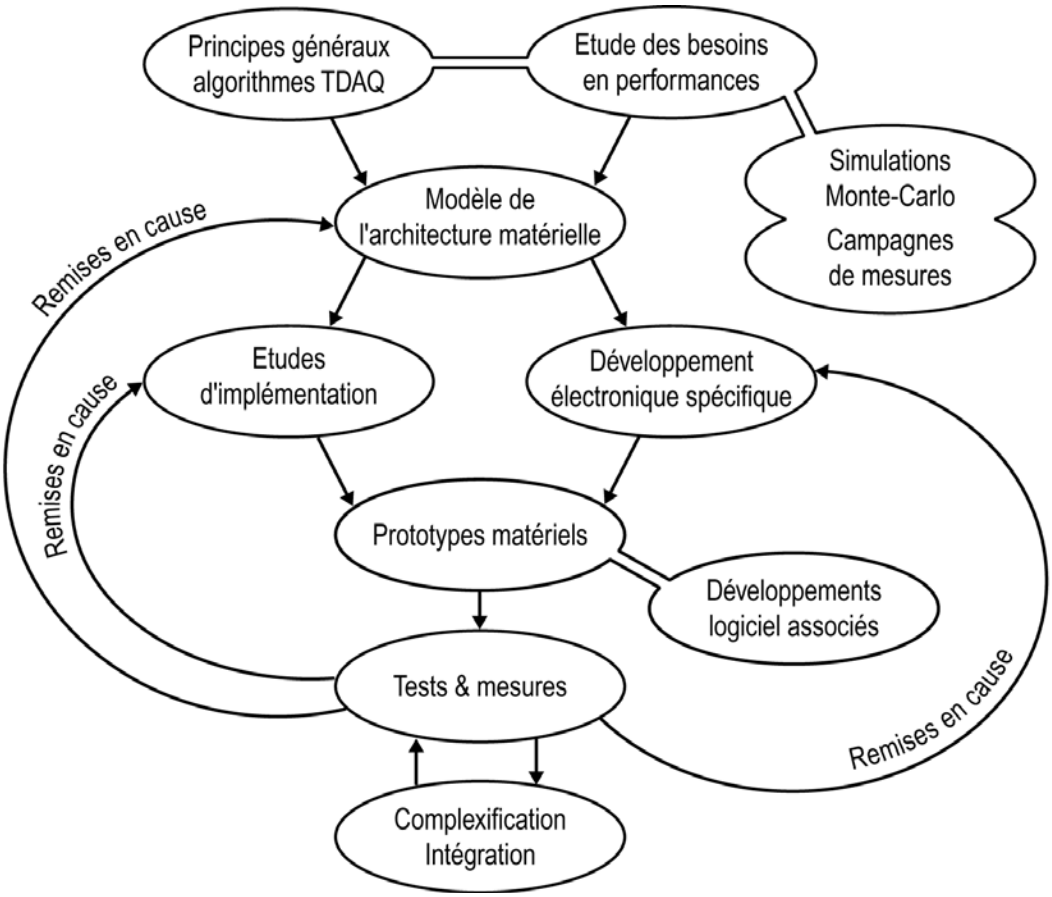


Fig. II-3 : Cycle de développement actuel d'un système TDAQ

Un modèle intuitif du système à base de diagrammes non spécifiques de type « *block diagrams* » est établi avec des spécifications générales sur les performances attendues multipliées par des facteurs de sécurité relativement importants. Les principes généraux des algorithmes de traitement et d'acquisition sont également énoncés. Tous ces développements conceptuels sont bien entendu régulièrement soumis à la collaboration* afin de minimiser les risques de méprise et donc d'erreurs de conception graves.

Ensuite, les ingénieurs vont étudier le marché pour commencer à se faire une idée sur quel type de solution (matérielle, logicielle, DSP, RISC, station de travail, standard réseau, électronique spécifique, etc.) adopter pour chaque classe de modules du sous-système. Puis, des tests vont commencer à être envisagés sur des plates-formes réelles réalisant des modèles très réduits et simplifiés des modules du système ; par exemple, l'équipe va se procurer quelques cartes processeurs munis de connexions réseau et de systèmes d'exploitation temps réel et commencer à effectuer des tests de bande passante dans différentes configurations simples. Des

algorithmes d'acquisition, de gestion de flots de données, de déclenchement sont élaborés et testés sur ces implémentations réduites, mais également, quand cela est possible, sur des simulations. Ces plates-formes de test sont progressivement complétées et développées, avec parfois des remises en cause de certains choix (matériels, architecturaux, etc.).

En ce qui concerne l'électronique spécifique, le développement de la partie dont on est certain qu'elle ne pourra se faire par logiciel (par exemple, l'électronique permettant de lire un ASIC* spécialisé) est lancé relativement tôt ainsi que ses spécifications d'interfaçage avec les autres modules. D'autres développements matériels sont éventuellement lancés pour tester leur potentiel de performance lorsque la faisabilité de la solution logicielle n'est pas évidente.

Du côté logiciel, en l'absence d'une méthodologie explicite, les ingénieurs ont du mal à élaborer une architecture logicielle de haut niveau, et le travail s'oriente souvent vers le développement de modules permettant d'effectuer des mesures simples sur les plates-formes de test, de réaliser le contrôle et l'initialisation des modules électroniques ou encore de tester des performances algorithmiques. L'architecture logicielle finale résulte le plus souvent d'un assemblage plus ou moins heureux de ces modules originels.

II.B.2 Manque de vision globale du logiciel

Le système complet résulte bien entendu de l'intégration progressive des différents modules logiciels et matériels développés, mais cette intégration se heurte souvent à des difficultés car les modules ont été développés par des personnes différentes, autour de l'électronique concernée.

Autrement dit, contrairement à l'élaboration de la partie matérielle (électronique) du système TDAQ, le développement du sous-système logiciel ne suit pas du tout une approche « *top-down* »*. Nous pensons que la raison principale de cet état de fait résulte d'une culture (au DAPNIA mais également dans la communauté HEP*) très orientée vers le matériel. Une fois l'architecture matérielle établie, celle-ci monopolise la vision du système dans les esprits de sorte que le logiciel n'est implicitement vu que comme des modules à développer autour de chaque composant matériel. Chacun développant relativement séparément le code lié à un composant donné, le logiciel ne bénéficie pas d'une vision aussi globale que

celle de l'électronique.

Dans l'expérience NA48, par exemple, le logiciel de contrôle de la carte de contrôle/émulation (carte MISC, cf. [B-8], pp. 97 à 98) du sous-système de déclenchement des chambres à dérive a été développé relativement indépendamment du logiciel du la ferme de calcul (cartes EW, cf. [B-8], pp. 101 à 115) et ce, bien que les deux logiciels soient en interaction étroite et tournent tous deux, dans leur première version, sur les mêmes processeurs DSP TMS320C40 [B-26]. La première conséquence de ces développements séparés fut une difficulté d'intégration logicielle accrue : beaucoup d'efforts supplémentaires furent dépensés pour faire fonctionner les deux logiciels en harmonie. La deuxième conséquence est une complexification inutile pour l'utilisateur du logiciel ; la procédure de configuration/initialisation, notamment, reste encore aujourd'hui entachée de difficultés malgré des efforts réitérés pour présenter une interface simplifiée à l'utilisateur.

Le logiciel des systèmes TDAQ est complexe, notamment parce qu'il concerne des systèmes distribués hétérogènes (Cf. II.A.6 et II.C.1). De plus, tout ou partie de ce logiciel est amené à être déployé sur de multiples plates-formes aussi bien au cours du développement que lors du déploiement final. Nous avons également vu au II.A.4 que nous avions besoin d'une version purement fonctionnelle du logiciel devant être intégrée dans les simulations de physique par Monte-Carlo. Généralement, cette version fonctionnelle du code est réécrite par des physiciens dans le langage de programmation de l'outil Monte-Carlo (FORTRAN et, plus récemment, C++). Dans ce cas, aucune procédure automatique ne permet de s'assurer de l'identité fonctionnelle du code réel et du code de simulation : les développeurs doivent s'en assurer « à la main » au prix d'un effort important.

II.C Evolution des systèmes TDAQ

Cette sous-section est consacrée à l'analyse des évolutions que connaissent les systèmes TDAQ aujourd'hui et des conséquences techniques afférentes. Cette analyse nous permet, d'une part, de souligner encore plus l'intérêt de disposer d'un cadre méthodologique spécifiquement dévolu à la conception/réalisation de tels systèmes, et d'aure part, de dégager les thèmes techniques qu'un tel cadre méthodologique doit traiter afin de répondre aux problèmes essentiels que rencontrent le développement des systèmes TDAQ.

II.C.1 Complexité, taille et donc parallélisme croissants

Ainsi qu'évoquée en introduction (section I), la tendance des systèmes de détection de la physique fondamentale est au gigantisme et à la complexification croissante. Il semble en effet qu'il existe une sorte de loi de rapport inverse entre les caractéristiques du phénomène observé et celles du détecteur dévolu à cette observation : plus ce qui est observé est petit et élémentaire, plus le détecteur utilisé doit être grand et complexe.

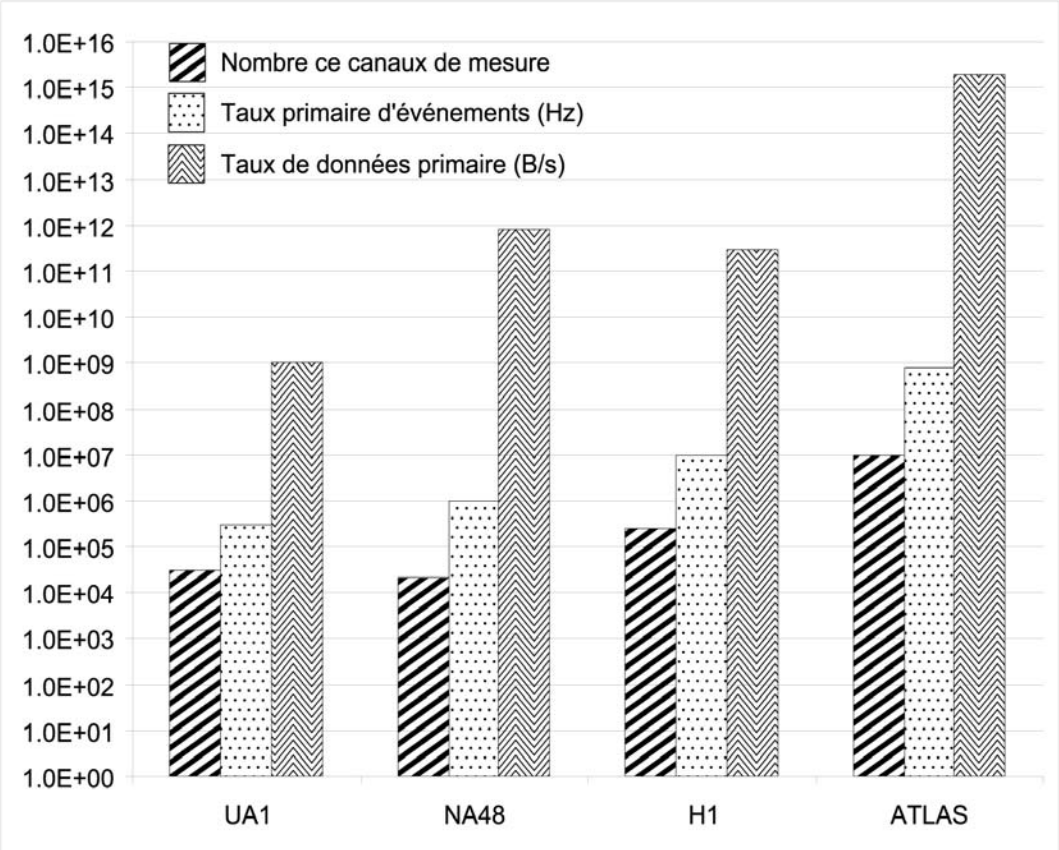


Fig. II-4 : Evolution typique des expériences de physique des hautes énergies en 2 décennies (échelle logarithmique)

La figure II-4 représente la progression chronologique des caractéristiques de taille et de complexité de 4 expériences typiques de physique des particules sur une période de 25 ans ([B-6] pp. 12 à 13, [B-8] p. 45, [B-3] pp. 3,8,12). L'échelle logarithmique de la figure montre bien qu'une expérience est plusieurs ordres de grandeurs plus grande et plus complexe que l'expérience de la génération précédente. Les expériences autour d'accélérateurs ne sont pas les seules à connaître une tendance au gigantisme. En effet, certaines expériences d'astroparticules

comme ANTARES [A-3], AMANDA [A-11], ou encore AUGER [A-16], qui s'intéressent à la détection de particules en provenance du cosmos, occupent des volumes très vastes. Les 400 nœuds de détection d'ANTARES, par exemple, s'étendent sur un volume de quelques 30 000 000 de m^3 (~300 m dans les 3 dimensions, cf. [B-2]) et les études actuelles sur son extension future envisagent un détecteur de 1 km^3 [B-10].

Complexité et taille croissantes sont ainsi des constantes des systèmes TDAQ, ce qui justifie que l'on recoure pour leur réalisation à des architectures de plus en plus massivement distribuées. Par nature, les systèmes TDAQ sont des systèmes distribués ne serait-ce que parce qu'ils sont constitués de modules géographiquement répartis. Comme nous l'avons vu en II.C.1, les expériences sur les télescopes ou les accélérateurs, peuvent s'étendre sur des dizaines, voire des centaines ou des milliers de mètres. D'autre part, la croissance exponentielle de paramètres tels le nombre de canaux de mesures (Cf. fig. II-4) dans les nouvelles expériences impose l'usage de systèmes fortement parallélisés afin de pouvoir traiter les flux de données associés[3]. Il s'agit donc d'assurer entre modules à l'intérieur des sous-systèmes ainsi qu'entre sous-systèmes, des mécanismes de communication simples, performants et, dans la mesure du possible, unifiés afin de réduire les efforts de développement et les coûts d'investissement. Ces mécanismes doivent également être adaptés aux différents types de parallélisme qu'imposent respectivement l'accroissement en taille des systèmes et l'augmentation des flots de données (Cf. IV.A.1).

II.C.2 Tendance à l'augmentation des contraintes de l'embarqué

Des défis techniques autres que le gigantisme et la complexité numérique se présentent à la recherche expérimentale en physique. Considérons à nouveau le cas des expériences ANTARES et AMANDA. Si le nombre de canaux de lectures dans ces expériences est bien moins important que dans les expériences sur collisionneurs (de l'ordre de 2000 canaux, voir [B-5]), elles connaissent en revanche des contraintes de fiabilité et de consommation bien plus importantes du fait de leur déploiement hors de portée de toute intervention humaine : les détecteurs d'ANTARES seront déployés à 2000 m sous la mer et ceux de AMANDA sous la glace du pôle sud. En ce qui concerne ANTARES, les ingénieurs espèrent qu'au

3. Même les systèmes embarqués sur les engins spatiaux sont le plus souvent distribués en raison de leur complexité mais aussi pour des questions de tolérance aux pannes.

bout de 3 ans de fonctionnement, environ la moitié des nœuds de détection seront encore en fonctionnement, ce qui est une contrainte très forte en termes de fiabilité par rapport à l'électronique industrielle courante. Aussi doit-on réduire au maximum la consommation des différents modules électroniques, afin de minimiser leur MTBF* (temps moyen avant panne). Par ailleurs, les modules étant situés à environ 2000 m sous la mer, les exigences de résistance à la pression des conteneurs autorisent peu de place, ce qui impose une intégration poussée des composants.

Les contraintes de l'embarqué se retrouvent également dans certaines parties des systèmes TDAQ sur collisionneur. En effet, la prochaine génération d'accélérateurs permettra d'atteindre des énergies telles que l'électronique proche des détecteurs sera soumise à des radiations. Elle devra non seulement employer de l'électronique durcie mais la présence de radiations interdira une présence humaine trop fréquente ce qui, dans la pratique, se traduira par une certaine inaccessibilité des modules. Cette inaccessibilité est également vraie de tous les modules enfouis dont l'accès nécessite un démontage long et fastidieux du détecteur. Enfin, d'une manière générale, le très grand nombre de modules impose des contraintes de fiabilité et de consommation importantes.

Ces contraintes, typiques des applications embarquées, vont bien entendu influer sur l'architecture système, les technologies logicielles employées ainsi que les exigences de qualité des composants matériels et logiciels. Les contraintes de consommation, par exemple, imposeront l'utilisation de processeurs basse consommation et donc peu puissants, ce qui aura un impact sur les technologies logicielles utilisables : contraintes sur les piles réseau, nécessité de tirer au maximum parti de ce qui peut être défini de manière statique dans les logiciels de déclenchement/acquisition, etc.

II.C.3 Evolution des langages et environnements logiciels

Il y a quelque 50 ans, le développement des appareillages pour les expériences de physique dans le domaine HEP* était essentiellement pris en charge par des physiciens, éventuellement assistés d'un certain nombre de techniciens. En d'autres termes, l'ingénierie électronique et informatique du système était pour l'essentiel le fait des physiciens. Ces physiciens étant également, bien entendu, ceux qui effectuent l'analyse scientifique des résultats d'expérience, il était naturel

qu'il existe un recouvrement (quand cela était possible) entre les techniques électroniques et informatiques employées pour les appareillages et celles mises en œuvre pour l'analyse.

Les expériences et leurs détecteurs prenant, au cours du temps, de plus en plus d'ampleur, les domaines relevant de l'ingénierie ont été progressivement pris en charge par des ingénieurs, même si ceux-ci étaient (et sont) encore souvent des physiciens qui se sont spécialisés dans un domaine technique. Cependant, en ce qui concerne l'activité scientifique proprement dite, à savoir l'analyse physique des résultats d'expériences, elle continue bien entendu à être une activité dévolue aux physiciens. Or, l'analyse physique repose essentiellement sur le traitement massif, et donc informatique, des données accumulées par les expériences et implique de la part du physicien une activité de programmation soutenue. Ainsi, les canevas* d'analyse tels PAW [A-25] et de simulation tel GEANT [A-24] développés au cours du temps par la communauté HEP résultent d'un travail impressionnant qui a également trouvé des applications dans d'autres domaines scientifiques [A-23]. De plus, les physiciens HEP ont toujours eu une vision de l'ensemble de leurs outils au niveau du code source et se sont toujours échangé les sources plutôt que les binaires, travaillant ainsi depuis longtemps selon un modèle de type « open source ».

Les physiciens sont donc obligatoirement conduits à connaître l'informatique, et exercent ainsi, par voie de conséquence, une influence directe sur le choix des technologies logicielles dans les systèmes TDAQ. C'est pourquoi, il nous a semblé nécessaire de donner un bref aperçu de l'usage et de l'évolution des technologies et des langages logiciels dans les expériences HEP au delà du strict domaine des systèmes TDAQ. Nous verrons, notamment au IV.B.4.5, que cela a son importance dans le processus de développement des systèmes TDAQ.

Jusqu'aux années 90, le langage de programmation exclusif employé par la communauté HEP a été le FORTRAN, langage de prédilection de toutes les activités de calcul numérique. Depuis, on constate une évolution de fond vers le C++ en raison du succès industriel de ce langage et des avantages reconnus apportés par la conception orientée objet. Ainsi, les différent outils logiciels de la communauté HEP ont été (ou sont) portés ou réécrits en C++, les exemples les plus connus étant ROOT [A-27] et GEANT 4 [A-26]. De même, les expériences de

nouvelle génération tels BABAR [A-17] au « Stanford Linear Accelerator » [A-9] ou encore ATLAS [A-12] ou CMS [A-13] au CERN* [A-1] ont décidé de baser l'ensemble de leurs développements informatiques sur le langage C++ et des canevas* orientés objets.

Cependant, il demeure dans la communauté des physiciens HEP une certaine réticence vis-à-vis de l'abandon du FORTRAN, réticence aggravée par la complexité du langage C++ et la difficulté conceptuelle de passer à une réflexion orientée objets. Certaines expériences difficiles avec des bases de données objet commerciales ont également contribué à associer dans l'esprit de nombreux physiciens le modèle objet en général et le langage C++ en particulier à de « belles constructions de l'esprit » inutilement complexes et peu adaptées au véritable besoin des physiciens qui est de « faire de la physique » en oubliant au maximum l'informatique. Ces réticences probablement légitimes sont à l'origine de développements mixtes FORTRAN/C++ laissant le choix du langage aux physiciens en enrobant le code FORTRAN de classes C++, ce qui est certainement très souhaitable du point de vue de la liberté d'action des physiciens mais probablement peu compatible avec des développements importants de qualité.

L'autre langage orienté objet qui fait son entrée dans le domaine HEP* est bien entendu le langage Java, dont la robustesse intrinsèque due à la gestion automatique de la mémoire est très certainement un atout non négligeable pour les développements HEP les plus lourds. L'usage de ce langage demeure cependant assez timide et reste pour l'instant cantonné aux interfaces graphiques et aux applications modestes.

La culture informatique dans le domaine HEP est donc d'un certain côté, familière avec des techniques de développement modernes tels les canevas* logiciels ou la programmation « open source » à grand nombre de programmeurs[4]. D'un autre côté, les échelles de temps dans les expériences de physique des hautes énergies se comptent en années voire en décades, ce qui entraîne une réactivité relativement limitée face aux nouvelles technologies.

4. *La communauté HEP a même été à l'origine du Web au sens où le protocole hypertexte HTTP a été inventé au CERN* par Tim Berners-Lee pour faciliter le partage des données entre laboratoires et centres de calculs scientifiques.*

II.D ENJEU : LA FAISABILITÉ DES SYSTÈMES DU FUTUR

Nous avons vu en introduction (section I) que le SEDI* se voit confier la charge de développement de nombreux systèmes TDAQ pour les expériences des trois physiques du DAPNIA*. Bien que les expériences soient techniquement très différentes, nous avons vu d'une part au II.A, qu'il est possible de dégager des points communs aussi bien structurels que fonctionnels entre leurs systèmes TDAQ et, d'autre part au II.B, que ceux-ci partagent peu ou prou des processus de développement analogues. En outre, l'évolution des systèmes TDAQ (II.C) vers une complexité accrue rend plus difficile la maîtrise du processus de développement de ces systèmes et donc la garantie de leur qualité. Il semble clair, au vu de ces évolutions, que sans la mise en œuvre d'outils au moins conceptuels et méthodologiques, cette maîtrise risque, à terme, de nous échapper en raison d'une complexité devenue ingérable. D'autre part, les budgets affectés aux expériences de physique ne peuvent en aucun cas connaître un rythme de croissance analogue à celui de la taille et de la complexité des instruments. Il est donc impératif de rationaliser nos processus de développement afin de dégager des gains en termes aussi bien d'intelligibilité des systèmes que de productivité des développements ; gains qui seuls, peuvent nous assurer la faisabilité des systèmes TDAQ du futur.

Ce travail de thèse constitue une proposition dans le sens d'une telle évolution des processus de développement. Cette proposition est constituée par un certain nombre de concepts et principes de développement originaux que nous tenterons d'expliciter et de formaliser. Il s'agira, à terme, d'en maximiser les gains en la matérialisant dans un outil informatique permettant une automatisation poussée des procédures exposées.

III

CANEVAS TDAQ : LES TECHNOLOGIES EMPLOYÉES

Cette section est consacrée à une analyse des technologies objet pertinentes pour nos problématiques de développement de systèmes TDAQ. L'essentiel de cette analyse est centré sur la question de la réutilisation logicielle.

III.A Le modèle objet : un paradigme incontournable

III.A.1 Succès du modèle objet dans l'informatique généraliste

Le succès et l'omniprésence du modèle objet dans l'industrie logicielle n'est plus à démontrer. Le signe le plus clair de ce succès est l'usage largement établi du langage C++ depuis le début des années 1990. Bien sûr, une grande part de l'activité informatique d'aujourd'hui concerne les développements liés à la croissance de l'Internet et des techniques associées et n'est plus réduite à l'élaboration de logiciels classiques dans un langage de programmation orienté objet tel le C++. Il n'en demeure pas moins que ces techniques utilisent toujours, pour une large part, le modèle objet, ne serait-ce que par l'usage du langage Java[1] ou des ORB* basés sur les normes CORBA* ou DCOM*. Il est également à noter que les technologies orientées objet sont d'autant plus utilisées que les projets sont de taille importante.

III.A.2 Adoption relativement récente dans le domaine temps réel

L'adoption du modèle objet dans le domaine du temps réel est moins évidente et est, en tout état de cause, bien plus lente que dans l'informatique généraliste. Bien entendu, les compilateurs C++ existent depuis longtemps pour les plate-formes destinées au temps réel comme les cartes VME* à base de DSP* ou RISC*, mais, si l'on se réfère au discours des vendeurs sur la demande de leurs clients dans le domaine durant les années 1990, il semble que jusqu'à récemment, les applications industrielles sur ces plate-formes aient été surtout réalisées en C. L'analyse des causes de cette inertie technologique sortant du cadre de cette thèse, nous nous contenterons d'en évoquer quelques unes possibles : l'infériorité réelle ou supposée du C++ par rapport au C en matière de performances, la crainte de se lancer dans une nouvelle technologie que l'on ne maîtrise pas pour des applications souvent critiques et exigeantes en matière de sûreté, ou encore l'absence de maturité des technologies orientées objet dans un marché très éclaté de fournisseurs de systèmes d'exploitation et de plate-formes électroniques. Plus spécifique-

1. *Notamment les Servlets [B-18] [B-17] et les Entreprise Java Beans [B-16].*

ment, dans le domaine HEP, les systèmes TDAQ dont le développement a débuté avant le milieu des années 1990 ont essentiellement utilisé le langage C ou même l'assembleur, témoin le code du sous-système de déclenchement des chambres à dérive de l'expérience NA48 [C-3].

En dépit de cette inertie technologique, l'usage du modèle objet et notamment du langage C++ (et dans une moindre mesure Java) finissent par s'imposer dans le domaine du temps réel, notamment grâce à l'explosion du marché de l'embarqué, essentiellement due au développement des objets nomades et des objets communicants. En effet, en raison de sa très grande évolutivité (voire volatilité), ce marché est très sensible au « *time-to-market* » et est donc très demandeur de technologies de réutilisation permettant de réduire ce paramètre, et les technologies objet ont, sur cette question, fait leurs preuves. Aussi, les outils, les environnements de développement et les *middlewares** proposés aujourd'hui par les vendeurs et les communautés *open source*, sont-ils presque tous basés sur des technologies et langages à objets[2] (voir par exemple [A-20]).

III.A.3 Qu'entend-on par le terme « objet » ?

Lorsque l'on parle de « modèle objet » ou de « langage orienté objet », on fait référence à un certain nombre de concepts de modélisation et de programmation. Deux aspects nous semblent centraux pour caractériser l'« orienté objet » : l'encapsulation et l'héritage. Le premier aspect correspond à la notion la plus intuitive que l'on a d'un « objet », à savoir une entité constituée de données structurées et de fonctionnalités associées à ces données. Cet aspect se rapproche de la notion de composant logiciel. Le deuxième aspect, plus abstrait, concerne les classes d'objet : il pose la possibilité entre classes, d'une part, d'une factorisation sémantique et d'autre part, d'un partage d'implémentation.

Par la spécification de relations d'héritage, on pourra donc factoriser en une seule classe A la sémantique commune à plusieurs classes C_1, C_2,..., C_n. On dira alors que « tout C_i est un A » ou que « tout C_i est une *spécialisation* de A » ou encore que « A est une *généralisation* des C_i ». Chaque C_i *hérite* des caractéristiques de A. Cet aspect peut être qualifié d'« héritage externe » au sens où les instances de C_i peuvent être extérieurement vues comme des instances de A et

2. *L'exception étant les systèmes d'exploitation, aussi bien généralistes que temps réel.*

pourront être employées partout où une instance de A peut être employée.

La même relation d'héritage permettra de spécifier un partage d'implémentation. En effet, chaque C_i pourra réutiliser directement toutes les méthodes implémentées par A comme si elles étaient les siennes, et toute modification de cette méthode dans A se répercutera automatiquement dans les classes C_i. Cet aspect peut être qualifié d'« héritage interne » au sens ou chaque C_i va reproduire au sein de sa propre structure interne, la structure interne d'une instance de A.

III.A.4 L'objet est une notion naturelle des systèmes répartis[3]

On commence à « faire de l'orienté objet » lorsqu'on ne voit plus un programme informatique comme un entremêlement d'appels de procédures agissants sur des blocs de données, mais plutôt comme un scénario particulier entre un ensemble d'objets offrant chacun un certain nombre de services. Cette vision favorise la perception psychologique qui tend à considérer un objet comme une entité douée d'une vie propre et d'une certaine autonomie même si, pour fonctionner, cette entité doit être intégrée au sein d'un ensemble correctement architecturé. C'est la raison pour laquelle, l'appel (ou invocation) d'une méthode sur un objet a pu être interprété comme un « envoi de message » ou une « requête d'exécution », et que la notion d'« objet actif » a pu se faire jour [C-11] [C-14]. En effet, à partir du moment où l'on perçoit le programme comme une collection d'objets qui se transmettent des requêtes d'exécution par appel de méthode, il devient concevable que ces appels puissent être asynchrones et qu'ainsi, l'exécution effective de la fonction se déroule dans un autre fil d'exécution que celui qui était l'hôte de l'appel. Chaque objet devient donc susceptible d'intégrer, en plus de sa structure de données et de ses méthodes, son (ses) propre(s) fil(s) d'exécution.

Grâce au paradigme objet, nous passons donc progressivement et naturellement d'une vision *monolithique* à une vision *distribuée* de l'application informatique, car une fois que les appels de méthode s'interprètent comme des requêtes d'exécution, rien n'oblige à ce que les messages qui véhiculent ces requêtes transitent dans le même espace d'adressage. En effet, aussi bien les mécanismes de communication inter-processus que les protocoles réseau sont là pour faire communiquer des entités n'appartenant pas au même espace d'adressage : il suffit que nos appels de

3. Dans tout l'ouvrage, les termes « réparti » et « distribué » sont synonymes.

méthodes emploient ces mécanismes pour que nous nous trouvions en présence d'une application proprement distribuée.

Ainsi, le paradigme objet, par son aspect d'encapsulation des structures de données et des fonctionnalités associées, suscite un cheminement psychologique qui mène naturellement à voir toute application comme étant – potentiellement – un système réparti.

*III.A.5 Emploi du formalisme UML**

A partir du moment où nous avons décidé de fonder notre cadre conceptuel de modélisation des systèmes TDAQ sur le paradigme objet, se pose la question de l'emploi d'un formalisme de modélisation objet. Nous pourrions nous contenter de nous exprimer en français, sans l'emploi d'un formalisme particulier, mais nous nous heurterions très vite à des problèmes de précision lors de la discussion sur tel ou tel aspect subtil de notre problématique. Pour surmonter ces problèmes, nous serions immanquablement amené à définir rigoureusement un formalisme minimal permettant d'exprimer les questions subtiles sans risque d'ambiguïté. Par conséquent, plutôt que de réinventer un formalisme *ad hoc*, nous décidons de recourir au seul formalisme objet normalisé qui constitue aujourd'hui un standard industriel de fait, à savoir le Langage de Modélisation Unifié : UML*.

UML [B-11] est un formalisme accompagné d'une notation graphique, dont la sémantique et la graphie sont normalisées par l'OMG* [A-19]. Il est inspiré des langages de modélisation de méthodologies orientées objet (notamment, OMT, Booch et OOSE). Il renferme l'essentiel des concepts qui fondent le paradigme objet et même plus. Outre son caractère de standard universellement reconnu, l'intérêt d'UML est qu'il ne constitue qu'un *langage* de modélisation indépendamment de toute méthodologie de développement (contrairement à OMT, Booch et OOSE, qui regroupent chacun une notation *et* une méthodologie). UML ne spécifie pas, par exemple, un ordre préférentiel pour l'usage des différents diagrammes de modélisation au cours d'un processus de développement. Libre à chacun, donc, d'utiliser le langage et la notation pour mettre en œuvre des méthodes ou des processus de développement particuliers, pourvu que la sémantique et la grammaire du langage soient respectées. De plus, afin que le formalisme soit adaptable à des domaines et des métiers différents, UML définit des mécanismes d'extension permettant aux concepteurs de méthodologie et d'AGL* de

préciser et de spécialiser les éléments de modélisation du langage. Ainsi, la norme ne donne qu'une définition ouverte de l'interprétation des différents éléments d'UML : les spécification de la norme sont plus des contraintes sémantiques que des définitions fermées. Le méta-modèle d'UML, c'est-à-dire le modèle qui définit les éléments de modélisation du langage ainsi que leurs articulations, définit différentes sortes d'*annotations* permettant aux utilisateurs d'étendre et de préciser les éléments du modèle sans pour autant être autorisés à créer de nouveaux éléments. Cet équilibre entre rigidité et permissivité rendent UML à la fois suffisamment souple pour pouvoir être adapté à des domaines différents et suffisamment contraint pour être doté d'une force expressive et d'une précision satisfaisantes.

Enfin, il est à noter que l'ensemble des vendeurs d'AGL orientés objet ont adopté ou adoptent la notation UML dans leur outil. Le mouvement de convergence est accentué par l'apparition et la normalisation du format XMI pour la représentation et l'échange de descriptions de modèles UML [B-19]. Par exemple, les AGL *Rational Rose* [C-13] ou *Objecteering* [B-20] sont aujourd'hui basés sur la notation UML. De plus, il s'est développé sur ces mêmes outils des plate-formes de développement spécialisées pour les applications temps réel : deux exemples pour les deux outils sus-cités sont respectivement *Rose RT*, développé par la société Rational et *ACCORD* [C-14] [C-15], développé par le LIST* au CEA*.

III.A.6 Travailler au niveau du méta-modèle

Au premier abord, UML est un langage qui permet la spécification à haut niveau de systèmes complexes, aussi bien dans leurs aspects structurels que dynamiques. Pour le concepteur d'applications, le premier intérêt de ce langage et de sa notation graphique est de lui permettre d'exprimer les spécifications du système sans s'embarrasser de détails de programmation qui, certes, sont nécessaires à la définition d'un système complet et fonctionnel, mais nuisent à l'intelligibilité et à la clarté du travail d'élaboration. Par rapport à un langage de programmation objet tel C++ ou Java, UML permet donc une *vision* plus synthétique et plus orientée vers l'architecture du système. Il se situe néanmoins au même niveau d'abstraction qu'un langage de programmation au sens où il permet de décrire la structuration et la dynamique du système à l'aide de primitives sémantiques plus ou moins paramétrées. Or l'analyse que nous nous proposons de faire des motifs récurrents* dans le processus de développement des systèmes TDAQ aboutira nécessairement à un

discours *sur* les mécanismes, contraintes et cheminements qui doivent guider ce processus : ces résultats ne seront donc formellement exprimables que dans un langage qui permette de désigner explicitement les éléments de modélisation utilisés par le concepteur. UML autorise ce discours « au niveau méta » puisque le méta-modèle d'UML est lui-même exprimable en UML. De plus, si nous voulons à terme pouvoir automatiser les motifs récurrents méthodologiques que nous aurons explicités, nous devrons recourir à un AGL permettant de les programmer, c'est-à-dire doté d'un langage de « méta-programmation ». Un tel langage nous permettra d'écrire du code susceptible d'interpréter les annotations UML et d'effectuer automatiquement les transformations de modèles correspondantes élaborées par notre travail. Nous reviendrons sur cette question en section V.

III.B TECHNIQUES DE RÉUTILISATION LOGICIELLE

Dès les débuts de leur histoire, les systèmes de traitement de l'information ont connu un rythme élevé d'accroissement en taille et en complexité. Au sein de l'activité informatique, la réutilisation logicielle s'est donc naturellement imposée comme une nécessité. La réutilisation logicielle peut être définie de manière très vaste comme l'utilisation, au cours d'un développement logiciel, de tout ou partie des fruits d'un travail déjà effectué lors de développements antérieurs, et ce, afin d'économiser du temps et de l'effort. Pratiquement, la réutilisation logicielle n'est réellement effective que si des efforts supplémentaires sont fournis lors des développements initiaux afin de rendre possible et aisée la réutilisation de leurs résultats. Cet effort supplémentaire correspond à un investissement initial dont on espère qu'il donnera lieu à des gains plus importants. La réutilisation logicielle n'a donc de sens que dans une perspective à moyen ou long terme, recouvrant des développements successifs dans le temps, et cela sera d'autant plus vrai que la part d'investissement initial liée au souci de réutilisation sera importante.

III.B.1 Langages procéduraux et les librairies logicielles

L'évolution des modèles de programmation a en grande partie été déterminée par le souci de réutilisation. Ainsi apparaît très vite la notion de fonction (ou procédure) qui s'accompagne d'un protocole de programmation (pile d'appels, adresse de retour, conventions de passage de paramètres,…) voire d'instructions spécialisées au niveau du processeur même (CALL, RETURN,…). La fonction permet la

programmation d'un code en sous-parties suffisamment indépendantes pour n'être écrites qu'une seule fois et réutilisées de multiples fois, d'abord dans le code lui-même et, ensuite, dans des codes ultérieurs. Les langages de programmation de haut niveau, tels FORTRAN ou C, qui sont apparus afin de faciliter l'écriture des programmes reposent en grande partie sur la notion de fonction, ce qui justifie l'appellation commune de « langages procéduraux ». Le besoin de réutilisation des fonctions d'un code à l'autre a engendré la notion de « librairie logicielle », ensemble de fonctions dont l'utilisation ne requiert en principe pas d'autres connaissances de la part du programmeur que leur fonctionnalité et leur signature. On désigne par l'abréviation API*, la description des fonctionnalités et des signatures d'une librairie logicielle.

Notons également la notion de « macro » apparue dans certains langages (et même dans les langages assembleur) qui permet le regroupement sous une expression concise et réutilisable de blocs de codes formellement identiques.

III.B.2 Réutilisation et langages objets

Outre leurs apports en termes d'encapsulation, de protection et de lisibilité, les langages objets ont permis une mise en œuvre plus poussée du souci de réutilisation. Alors qu'une librairie procédurale ne fournit qu'un ensemble de fonctions destinées à être appelées (éventuellement par des fonctions d'une autre librairie), les fonctionnalités d'une librairie d'objets peuvent non seulement être utilisées telles quelles par instanciation des objets de la librairie, mais également étendues par les mécanismes d'héritage et de redéfinition[4]. De plus, les appels de méthodes au sein des objets de la librairie peuvent être naturellement redirigés vers un objet utilisateur par les mécanismes polymorphiques : l'ensemble des objets réutilisables n'est donc plus réduit aux objets appelés mais peut également comprendre des objets appelants, ce qui rend possible de nouvelles architectures logicielles et annonce les nouveaux édifices dédiés à la réutilisation logicielle que sont les canevas*.

III.B.3 Réutilisation et temps réel

La réutilisation dans les développements logiciels temps réel inclut des problématiques spécifiques supplémentaires. Il ne s'agit pas seulement de réutiliser les

4. En anglais « overriding ».

différents modules ou objets mais également la structure d'exécution et d'interaction de ces modules et objets ainsi que les stratégies d'ordonnancement, de synchronisation, etc. Or, ces notions spécifiquement temps réel ne trouvent pas leur expression dans le modèle objet classique [C-14]. La réutilisation logicielle dans le domaine temps réel doit donc faire appel à des modèles conceptuels qui étendent le modèle objet classique aux problématiques propres aux notions temps réel. Le modèle de l'objet actif compte parmi les extensions conceptuelles les plus fructueuses du modèle objet permettant d'introduire naturellement la propriété de concurrence (Cf. III.A.4, [C-14] [C-11]) : chaque objet étant susceptible de porter sa (ou ses) propre(s) tâche(s), il devient possible de *séparer* l'invocation de méthode de son mode d'exécution, et donc de *réutiliser séparément* (et de manière indépendante) le code proprement fonctionnel et le code régissant le modèle d'exécution[5]. Ce fonctionnement n'est pas sans rappeler les mécanismes de réflexion — même si dans un outil de type canevas, il doit être mis en œuvre de manière plus explicite (Cf. par exemple [B-22]). En séparant la spécification du modèle d'exécution du modèle proprement applicatif, on améliore la réutilisabilité de chacun. Par exemple, le portage de l'application complète sur un système d'exploitation dont les API* de concurrence sont très différentes[6] ne nécessitera des modifications importantes que dans le code concernant le modèle d'exécution, assurant de ce fait une intégrité importante du modèle applicatif. Dans l'approche ACCORD ([C-14] [C-15]), les appels de méthodes des objets actifs sont automatiquement traduits en une requête qui sera traitée en soi, et selon ses paramètres de contraintes temps réel, par un ordonnanceur totalement séparé du reste de l'application : on pourra donc modifier la logique ou l'implémentation de l'ordonnanceur et réutiliser tel quel le reste de l'application.

Malgré cela, la réutilisation de classes d'objets actifs rencontre des difficultés, notamment en ce qui concerne la question de l'héritage des spécifications du modèle d'exécution. Qu'est-ce qui, dans les spécifications du modèle d'exécution, doit traverser les liens de spécialisation ? Aucune réponse ne fait encore l'unanimité.

5. *Le modèle d'exécution comprend, entre autres, les spécifications de concurrences (priorités, échéances,...) et d'intégrité de fonctionnement (conformité à une machine d'états).*
6. *Exemple : portage d'une application depuis Linux vers VxWorks.*

III.B.4 Les motifs récurrents

Au delà de la seule réutilisation de code ou de composants logiciels, l'idée de réutilisation « architecturale » est apparue dans la recherche et l'industrie informatiques sous la forme des *patterns*[*], que nous traduisons ici par l'expression « motifs récurrents » (la justification de cette traduction deviendra claire dans ce qui suit). Le concept de « *pattern* » –ou « motif récurrent », donc– a été directement inspiré des travaux de Christopher Alexander sur les architectures de bâtiments [C-37] dont l'idée maîtresse est d'envisager la possibilité de découvrir des structures ou principes de construction qui répondent *objectivement* à des besoins humains en matière de qualité fonctionnelle et de confort psychologique. Dans ce contexte, un motif récurrent correspond à une solution architecturale *éprouvée* répondant à une problématique *récurrente* dans la construction de bâtiments. Alexander propose donc que l'ensemble de la profession concoure à l'établissement d'un *catalogue* de telles solutions sous une forme homogène afin que la communauté des architectes bénéficie directement des bonnes pratiques nées de l'expérience et, pour ainsi dire, *ne réinvente pas la roue* à chaque génération.

L'application de cette idée à l'architecture logicielle a été reprise par la recherche informatique à la suite de l'article fondateur de Beck et Cunningham [C-38] et popularisée par l'ouvrage de Gamma, Helms, Johnson et Vlissides[7] [C-39]. En informatique –et plus particulièrement en informatique orientée objets– un motif récurrent est une micro-architecture type (ou élément architectural) de composants en interaction apportant une réponse élégante, intelligible et réutilisable à un problème type. L'explicitation des motifs récurrents par les hommes de l'art est ainsi considérée comme une contribution efficace à l'amélioration de la qualité logicielle en ce qu'elle promeut l'extensibilité, la flexibilité et la réutilisabilité des programmes informatiques.

III.B.4.1 Exemples

Le motif « *Observer* » par exemple, exposé dans [C-39], pp. 293 à 303, est à la base de l'architecture des applications graphiques où le même objet document est visualisé et modifié à travers des vues multiples. Il constitue, en d'autres termes, le cœur de l'architecture « *Model-View-Controller* » reprise sous des formes diverses

7. *Dans la littérature sur les motifs récurrents, ces auteurs sont souvent référencés par l'expression « La bande des quatre » (Gang of four).*

dans presque tous les environnements de développement d'interfaces graphiques[8] (Cf. [C-43]). Il permet une réutilisabilité et une maintenabilité accrues grâce au découplage qu'il institue entre le modèle intrinsèque d'un objet document et ses multiples représentations et modes d'accès graphiques possibles.

Un autre exemple, plus proche des problématiques de distribution et donc de nos préoccupations liées aux systèmes TDAQ, est le motif « *Acceptor and Connector* », exposé dans [C-12]. Il s'agit, dans ce motif, de découpler au sein d'une connexion client-serveur, les phases d'initialisation et de traitement du service en les explicitant sous forme d'instances de classes bien distinctes. Par exemple, l'acceptation d'une connexion client dans un serveur va être prise en charge par un objet de classe `Acceptor` qui créera et lancera ensuite l'objet de classe `ServiceHandler` en charge de traiter le service proprement dit. Ainsi l'objet `:ServiceHandler` n'a aucune connaissance de l'objet `:Acceptor` et effectue son service sous l'hypothèse que toutes les initialisations nécessaires ont été effectuées. Ce motif permet de faire évoluer le code lié au traitement du service indépendamment du code d'initialisation et d'établissement d'une connexion ; en particulier, il permet de réutiliser directement ce dernier pour des services de natures très différentes. L'expérience prouve que les stratégies d'initialisation dépendent de critères (latences de connexions, cartographie et configuration système, etc.) qui sont en général déconnectés des critères de performances spécifiquement liés au fonctionnement du service ; aussi, une organisation qui doit souvent mettre sur pied des architectures des services distribués peut-elle grandement tirer profit du motif « *Acceptor and Connector* » afin de se constituer des librairies de composants d'initialisation et de services indépendantes et composables entre elles. Le *middlewar*e de développement d'applications à objets distribués C++ ACE-TAO, par exemple, a mis ce motif en œuvre avec succès ([B-31], [A-20]).

III.B.4.2 Pallier l'insuffisance des mécanismes objets

Les mécanismes de réutilisation liés aux langages objets que nous avons évoqués au III.B.2 demeurent insuffisants à eux seuls pour permettre une réutilisation efficace des structures complexes. Selon [C-44], la simple spécification des opéra-

8. Par exemple, les libraires d'interface graphique du langage Java tels java.awt ou javax.swing, reprennent explicitement le motif « Observer » et l'architecture « Model-View-Controller ».

tions et des attributs permise par un langage objet est insuffisante pour définir le rôle d'une classe au sein d'un élément architectural, même dans le cadre plus riche d'un « *template* », c'est-à-dire une déclaration de classe (voire de collaboration) paramétrée équivalente des « macros » des langages procéduraux. Cette insuffisance est palliée par les motifs récurrents car ceux-ci permettent de spécifier en plus *comment s'organisent* les appels d'opérations, mise à jour d'attributs et autres actions permettant d'effectuer une tâche particulière. De manière analogue, la mise en œuvre d'un motif récurrent relève plus de la notion UML de « *realization* » ([B-12], p. 2-18) que de celle de « *binding* » ([B-12], pp. 2-225, 2-226) ; il s'agit en effet d'instancier une architecture d'objets respectant des contraintes et comportements suffisamment complexes pour ne pas pouvoir être exprimée par un énoncé paramétré statique. Ainsi, les motifs récurrents sont-ils vus comme les compléments indispensables des langages objets à partir du moment où l'on s'attaque à des réalisations réelles complexes. D'ailleurs l'une des critiques[9] faites à l'encontre des motifs récurrents de la « bande des 4 » ([C-39]) est que certains de ces motifs (comme le motif « *Iterator* ») constituent justement des « béquilles » servant à compenser certaines insuffisances conceptuelles du C++.

III.B.4.3 Formalisation des motifs récurrents

Afin de répondre à des exigences de clarté et de précision, les auteurs de motifs récurrents poussent à une certaine formalisation dans la présentation des motifs, ce qui a motivé l'apparition des motifs pour l'écriture de motifs ([C-40], [C-41]) qui s'appuient, entre autres, sur la spécification de rubriques standardisées à renseigner : nom évocateur, présentation systématique des forces en jeu, conditions d'applicabilité, etc. La norme UML prévoit à cet égard, une représentation des motifs récurrents en y associant même une notation spécifique ([B-12], pp. 2-116, pp. 3-118 à 3-122), mais elle précise bien que cette représentation n'englobe pas toutes les caractéristiques sémantiques d'un motif récurrent. Certains auteurs tels ceux de [C-44] critiquent cette représentation en en dénonçant le caractère incomplet voire réducteur qui limite essentiellement le motif à une collaboration paramétrée.

---

9. Notamment évoquée lors d'une session spéciale humoristique de la conférence OOPSLA'99 intitulée « Le procès de la bande des 4 ». Cf. http://w3.one.net/~swessels/pages/steve/squeak/oopsla99report.html et http://www.acm.org/sigs/sigplan/oopsla/oopsla99/2_ap/tech/2d1a_gang.html. Les principaux « détracteurs » des motifs récurrents étaient issus de la communauté du langage Smalltalk.

D'une manière générale, la littérature des motifs récurrents ([B-36], [C-40]) est relativement unanime sur l'idée que les motifs récurrents sont avant tout destinés à transmettre de manière *pragmatique* la *sémantique* architecturale de solutions à des problèmes de conception et qu'à ce titre, on ne pourra jamais les formaliser de manière stricte et ce, quel que soit le langage utilisé. Pourtant, certains acteurs de la recherche ([C-44], [C-15], [C-14]) ou de l'industrie ([B-20], [A-28]) envisagent sérieusement la possibilité de *programmer* des motifs récurrents afin de permettre leur intégration concrète au sein d'outils informatiques d'aide au génie logiciel. Une telle programmation semble *a priori* en contradiction avec le discours très « orienté sémantique » qui est à l'origine des motifs récurrents. On peut cependant lever cette contradiction et concilier la vision classique des descriptions de motifs récurrents avec leur programmation en considérant celle-ci comme une forme d'implémentation généraliste ou générique du motif. Cependant, pour éviter de retomber à nouveau dans une représentation trop réductrice des motifs récurrents (telle celle de collaboration paramétrée), les auteurs préconisent l'usage d'un langage de manipulation de méta-modèle pour programmer les motifs. L'idée qui préside à cette proposition est qu'un motif récurrent peut le plus souvent être traduit par la *transformation* d'un modèle selon les recommandations du motif. Par exemple, le motif « *Proxy* » (Cf. [C-39], pp. 207 à 217) peut tout-à-fait faire l'objet d'une telle transformation au sens où l'on peut envisager une génération au moins partiellement automatique d'un modèle structurel et comportemental de la classe *Proxy* à partir d'un modèle d'interface[10].

Nous verrons au IV.C que notre canevas TDAQ reposera en grande partie sur la possibilité de formaliser des motifs récurrents afin d'automatiser au maximum le travail des développeurs. Nous défendrons en effet l'idée que les éléments architecturaux fréquemment rencontrés au cours des développements TDAQ doivent pouvoir être exprimés sous la forme de motifs récurrents et automatisés par des programmes de transformation de modèles idoines.

III.B.5 Les canevas

Nous avons vu au III.B.1 que l'une des formes les plus classiques de réutilisation

10. C'est ce que fait, à un niveau assez modeste, un compilateur IDL CORBA, en générant le code associé au proxy à partir d'une définition d'interface en IDL.Cf. http://www.omg.org/gettingstarted/omg_idl.htm

est la librairie logicielle. En réalité, l'expérience prouve ([C-19]) qu'un composant isolé est rarement réutilisable tel quel : celui qui l'a produit le place implicitement ou explicitement au sein d'un « canevas »* (en anglais *framework*), c'est-à-dire d'un ensemble de composants articulés selon une architecture logicielle précise sans laquelle la réutilisation du composant est impossible. Autrement dit, les librairies qui atteignent réellement leur but de réutilisation sont en fait des canevas.

Selon [C-20], un canevas est constitué d'une hiérarchie de classes rattachée à des modèles d'interaction de manière à produire des applications complètes à travers différentes sortes de spécialisations ; en ce sens, les canevas permettent la réutilisation d'un code *et* d'une architecture. Beaucoup d'auteurs définissent en effet le canevas comme une « application semi-complète » ([C-22]) que le développeur n'a plus qu'à compléter par des mécanismes de spécialisation, composition et configuration afin d'obtenir une application en ordre de marche. L'exemple le plus classique de canevas cité dans la littérature est celui de canevas de développement d'applications graphiques tel l'environnement de développement « Visual Studio » de la société Microsoft ou la librairie « java.awt » du langage Java de la société Sun : ces canevas mettent en œuvre une architecture (boucle d'événements, rafraîchissement par *callbacks*, etc.) et des composants (boutons, boîtes de dialogue, etc.) permettant de développer des applications complètes par spécialisation, composition et configuration.

Cette vision signifie qu'un canevas représente le facteur commun, sous forme de code concret, d'une classe d'applications fondée sur une architecture et des composants similaires. La conception d'un canevas passe donc par la mise en évidence des structures invariantes au sein d'une classe d'applications et par leur réification sous forme d'un modèle de classes ou de composants ([C-21]). Inversement, il s'agit de réifier également les zones de variabilité maximale afin de présenter aux développeur des points d'entrée bien déterminés pour la production de code ou la spécification de configuration particulières à l'application instanciée. Ces zones de variabilité sont appelés les « points chauds » (*hot spots*) du canevas. Leur réification aboutit le plus souvent à un point d'insertion de composant (*hook*) ou une classe abstraite, référence polymorphique vers les classes concrètes destinées à implémenter le point chaud. Au bout du compte, un bon canevas cède le contrôle des points chauds au développeur et lui retire le contrôle des structures invariantes.

III.B.5.1 Canevas et motifs récurrents

L'idée de canevas prolonge naturellement la notion de motif récurrent. A l'instar des motifs récurrents, il s'agit pour les canevas de mettre en œuvre une forme de réutilisation architecturale. Ainsi, Les canevas et les motifs récurrents sont souvent intimement associés dans la littérature ([C-22], [C-23], [C-25]), bien que correspondant à des concepts différents. En effet, un motif récurrent représente un *élément architectural* et non une architecture complète et recouvre une description plutôt abstraite et « quintessentielle » de cet élément. Inversement, un canevas met le plus souvent en œuvre une architecture complète susceptible de produire des applications, et ce, sous forme de modèle concret, voire de code et de composants logiciels. Reprenant la formulation de [C-22], nous pouvons dire que les canevas et les motifs récurrents sont tous deux des moyens de faire de la réutilisation à grande échelle en explicitant les stratégies éprouvées de développement logiciel dans un contexte particulier ; leur différence principale réside en ce que les canevas sont orientés vers la réutilisation de conceptions détaillées, d'algorithmes et de code, alors que les motifs relèvent plutôt de la réutilisation de thèmes de conception, même si le détail de l'implémentation de ces thèmes peut ne pas être pas directement réutilisable. Les canevas sont souvent constitués par la mise en œuvre concrète de nombreux motifs récurrents ([C-40], [C-22], [C-16], [B-31]). A cet égard, le motif « *System of Patterns* » de [C-40] représente un canevas comme un « système de motifs récurrents » et fournit des classes d'architectures et une méthodologie de fabrication concrète de canevas à partir de motifs récurrents.

III.B.5.2 Classifications de canevas

Nous avons relevé dans la littérature ([C-10], [C-21], [C-17], [C-24]) deux grandes manières de classifier les canevas : la classification par cible et la classification par structure. Nous allons brièvement les décrire afin de voir comment nous comptons situer notre propre canevas TDAQ.

III.B.5.2.a Classification par cible

Les canevas peuvent être distingués selon leur « cible », c'est-à-dire selon la couche logicielle visée. Nous pouvons alors les placer dans deux grands ensembles :

1. Les canevas d'infrastructure système et d'intégration *middleware* *.

Cet ensemble regroupe les canevas destinés à produire un aspect techniquement spécialisé mais qui concernent presque toutes les applications. Il en est ainsi des canevas de développement d'interfaces graphiques, des *middlewares** de communication tels les implémentations d'ORB* ou les exécutifs de communication par messages, des plates-formes de développement de SGBD, etc. Ces canevas réalisent une architecture de composants particulière, bien adaptée à la problématique informatique traitée (distribution, interface homme-machine, etc.). Leur domaine d'application est très vaste car ils traitent de problématiques informatiques horizontales qui traversent tous les domaines métier. L'utilisation simultanée de plusieurs de ces canevas au sein de la même application peut causer des problèmes difficiles de composition de canevas (Cf. [C-18]).

2. Les canevas métiers et d'applications d'entreprise.

Un canevas appartenant à cet ensemble est plus conforme à la définition donnée en page III-15 : c'est une application semi-complète dédiée à un domaine métier ou à une entreprise particulière. Un tel canevas résulte d'une analyse de domaine et de la refactorisation de plusieurs applications afin d'en dégager l'architecture commune ([C-17], [C-9]). L'entreprise ou l'organisation détentrice peut ainsi produire de nouvelles applications liées à son domaine par spécialisation, composition et configuration du canevas. De nombreuses organisations telles des banques, des industriels des télécoms ou des avionneurs ont développé ainsi des canevas spécialisés leur permettant d'améliorer leur productivité de développement (Cf. [C-8], [C-9]). Ces canevas concernent un domaine plus restreint car plus spécialisé et sont donc *a priori* globalement plus coûteux ; leur développement ne se justifie que si de multiples développements applicatifs dans le même métier sont à prévoir.

III.B.5.2.b Classification par structure

Cette classification repose sur la structuration du canevas, structuration qui va déterminer son mode d'usage, c'est-à-dire la manière dont les développeurs vont pouvoir produire des applications à partir du canevas. On distingue alors essentiellement deux types de canevas :

1. Les canevas « boîte blanche »

Ce sont les canevas dont on se sert par spécialisation de classes particulières ou par

la mise en œuvre d'une architecture de motifs récurrents. Un canevas boîte blanche est ainsi une véritable application « incomplète » au sens où le développeur sera amené à en compléter le code source en y adjoignant son propre sous-système par la réalisation de classes spécialisées dérivant des points chauds du canevas. C'est un mode de travail basé sur la *délégation*. Un tel canevas nécessite de la part du développeur une bonne connaissance de son architecture générale et de son fonctionnement interne ; il est donc difficile à utiliser mais il présente plus de souplesse et permet de produire des applications mieux adaptées et plus performantes ([C-21]).

2. Les canevas « boîte noire »

Ils se situent à l'opposé des canevas boîte blanche car ils s'utilisent par composition, paramétrage et configuration des composants du canevas. Un canevas boîte noire ne nécessite pas de la part du développeur une connaissance approfondie de l'architecture interne de l'application produite. Il présente au développeur des menus de composants, au pire des classes paramétrées, comme autant d'alternatives à la spécification d'un point chaud. Celui-ci devra faire des choix de configuration afin de d'instancier l'application souhaitée : c'est un mode de travail par *composition*. Un canevas boîte noire est donc plus facile à utiliser mais plus rigide ; de plus, son développement est plus ardu car les concepteurs du canevas doivent prévoir le maximum de cas possibles tout en évitant le développement de composants inutiles ([C-21]).

Les canevas réels se situent en réalité quelque part dans un continuum entre boîte blanche et boîte noire. De plus, un canevas est un produit trop complexe pour pouvoir être construit simplement à partir d'analyses de domaine ou d'études de cas ([C-9], [C-10]). Le plus souvent, un canevas commence comme boîte blanche par refactorisation de plusieurs applications, puis évolue par itérations successives vers un canevas boîte noire au fur et à mesure que les composants éprouvés s'accumulent et que les frontières du domaine deviennent plus nettes dans l'esprit des concepteurs ([C-21], [C-23]).

III.B.5.3 Quel canevas pour les systèmes TDAQ ?

Les avantages attendus du développement d'un canevas métier ne se réduisent pas aux seuls aspects techniques (réutilisabilité, maintenabilité, robustesse, interopérabilité, etc.). L'expérience montre ([C-8], [C-9]) que la fabrication réussie d'un

canevas métier implique une clarification et une rationalisation de l'organisation du développement qui induit, entre autres, des améliorations économiques et managériales non négligeables. Aussi est-il particulièrement intéressant d'envisager le développement d'un canevas TDAQ, spécialement en ces temps de réductions budgétaires pour les sciences fondamentales.

La question est de savoir de quelle nature doit être un tel canevas. Au vu des sous-sections précédentes, nous pouvons commencer à répondre à cette question, réponses qui seront complétées dans la sous-section III.C. Il semble clair qu'un canevas TDAQ est un canevas métier, même si pour concrétiser les applications, la composition avec des canevas *middlewares* semble inévitable (par exemple pour les problématiques de distribution). De plus, la grande variété des systèmes TDAQ (Cf. II.A) exige des outils de développement souples ; aussi, tout canevas TDAQ doit-t-il essentiellement être de nature « boîte blanche », même si l'on peut y envisager l'intégration progressive de composants directement réutilisables, notamment pour la gestion des problématiques suffisamment générales comme celles évoquées au II.A (par exemple l'assemblage d'événements tel que défini au II.A.3). Mais nous verrons plus loin qu'en y adjoignant une composante méthodologique, nous pourrons traiter de manière plus satisfaisante aussi bien la réalisation d'un tel canevas que son adéquation à une activité concrète de développement de système TDAQ.

III.C Quelle approche pour le développement TDAQ ?

III.C.1 Critères de choix technologiques

L'industrie et la recherche logicielles fournissent déjà différentes classes de solutions aux problématiques de développement de systèmes. Il s'agit donc pour nous d'en retenir ce qui nous semble praticable et d'élaborer, dans le cadre choisi, une solution adaptée à la conception, l'implémentation* et la mise en œuvre de systèmes TDAQ pour la physique. Notre analyse du domaine en II.A, II.B et II.C indique clairement que, concernant la *structure* des systèmes TDAQ, nous devons avant tout nous intéresser aux technologies liées aux systèmes distribués, temps réel et, dans certains cas, embarqués ; nous devrons ainsi étudier l'applicabilité de ces technologies au sein d'un processus de développement fondé, lui, sur des technologies de génie logiciel, c'est-à-dire des technologies permettant de traiter la

dynamique de développement de ces mêmes systèmes.

L'industrie et la recherche informatiques proposent de nombreuses approches pour la spécification, la conception et la réalisation de systèmes distribués, temps réel ou embarqués. L'un de nos critères majeurs pour adopter une approche plutôt qu'une autre sera le degré de maîtrise que nous pourrons avoir sur celle-ci ainsi que le caractère plus ou moins avéré de standard industriel des technologies associées. Par exemple, une approche fondée sur CORBA* aura *a priori* notre faveur car CORBA est une norme bien établie et constitue un standard industriel dont l'évolution cohérente à moyen terme semble assurée. Bien entendu, si nous nous interdisons de fonder notre approche sur un formalisme ou un outil non standard dont nous n'aurions pas la maîtrise, nous nous autorisons en revanche l'usage ou la reprise des concepts et idées afférents. Nous pouvons ainsi distinguer les critères suivants, à appliquer lors des choix technologiques :

1. Favoriser l'usage de langages, notations et normalisations qui soient des standards industriels.
2. Eviter l'usage de bibliothèques non standard dont le code source n'est pas disponible.
3. Favoriser l'usage d'outils et de langages multi-plates-formes.
4. Eviter autant que possible l'usage d'outils qui imposent des formats propriétaires, et, dans les cas où on ne peut l'éviter, s'assurer de l'existence d'outils de transformation en formats standards.
5. Favoriser les outils méthodologiques et procédés de développement incitant à rationaliser le développement d'un système par l'incitation à la conception modulaire, à l'abstraction et à la factorisation des différents aspects du système.
6. *In fine*, automatiser les procédures de développement afin de dégager de véritables gains de productivité.

III.C.2 Systèmes distribués : les approches possibles

De nombreuses approches ont été proposées par l'industrie et la recherche pour aborder la question du parallélisme et des systèmes distribués. On peut sommairement diviser ces approches en trois catégories, présentées de manière détaillée dans [C-24] : les approches applicative, intégrée et réflexive. L'approche qualifiée

d'« applicative » consiste essentiellement à développer un canevas de développement d'applications distribuées. L'approche intégrée consisterait à *unifier* les paradigmes de programmation séquentielle des langages de programmation (notamment le paradigme objet) avec les paradigmes liés au parallélisme et aux systèmes concurrents en général. Cette unification s'exprimerait par l'établissement d'un *langage* de programmation qui *intégrerait* l'ensemble de ces paradigmes[11]. Enfin, l'approche réflexive se fonderait sur un mode de programmation qui intégrerait les modèles d'exécution et de communication aux modèles structurels « utilisateur » en un *même langage*, tout en les confinant dans des niveaux séparés : le premier niveau (le niveau « utilisateur ») comprendrait la description structurelle de l'application ainsi que ses fonctionnalités ; ce serait, pour ainsi dire, une description « naïve » de l'application sans aucune considération sur ses modes d'exécution et de communication inter-objets. Ce premier niveau serait assujetti à un niveau « méta » qui comprendrait la description des modèles d'exécution et de communication sous la forme d'objets assurant les tâches d'ordonnancement, de transmission, etc. et ce, indépendamment des descriptions purement applicatives du niveau utilisateur.

III.C.3 L'approche « canevas » comme seule praticable

Les critères de maîtrise que nous avons décrits en III.C.1 semblent éliminer de notre travail les approches intégrées et réflexives à moins que nous ne décidions de développer nous-mêmes les langages évolués nécessaires à ces approches ou de participer activement à leur développement. Mais l'invention (ou même la promotion) de nouveaux langages ne semble pas constituer une approche réaliste dans le cadre d'une activité de développement de systèmes réels soumis à des impératifs de pérennité forts. Un langage « maison » aurait peu de chances de s'imposer réellement dans les habitudes de développement d'une organisation n'ayant pas les moyens de promouvoir ce langage au moins au niveau du domaine métier. De plus, la maintenance d'un langage de niveau *industriel* (y compris son adéquation à l'évolution des plates-formes matérielles et logicielles) représente une charge qui se situe en dehors des capacités d'un laboratoire du milieu HEP* tel le DAPNIA[12].

11. L'intégration profonde de constructions comme la classe « Thread » ou l'interface « Runnable » dans le langage Java autorise, dans une certaine mesure, à considérer ce langage comme une réalisation, au moins partielle, de l'approche intégrée.

En outre – et ceci est également souligné par les auteurs de [C-24] – l'approche intégrée réduit inévitablement le champ des possibles en matière de modèles d'exécution et de communication. En effet, en voulant les intégrer dans un langage de programmation au même niveau que le langage de description structurelle et fonctionnelle de l'application, on est immanquablement amené à faire le choix d'un point de vue particulier en la matière, ce qui, au mieux, risque de poser des problèmes d'efficacité dans les cas où le modèle d'exécution mis en œuvre est peu compatible avec la philosophie de l'approche en question.

Il est certain que l'approche réflexive résout ce genre problème car, ne faisant aucun choix en matière de modèles d'exécution et de communication, elle laisse le programmeur libre de mettre en œuvre sa propre politique au niveau « méta ». En effet, la spécificité du langage associé (ainsi que, le plus souvent, la machine virtuelle ou l'exécutif) se limite à l'articulation des niveaux « utilisateur » et « méta » et ne met pas en œuvre un modèle d'exécution ou de communication particulier : il se contente de « passer la main » au niveau « méta » lorsque des décisions concernant la communication ou l'ordonnancement doivent être prises, laissant le développeur libre d'y programmer l'algorithme de décision qu'il souhaite. Il n'en demeure pas moins qu'une telle approche nécessite, sinon un langage (tel ABCL/R dans [C-27] et [C-26]), du moins un système d'exploitation ou une machine virtuelle intégrant les mécanismes de réflexion (comme dans [B-21] et [C-27]). Encore une fois, la maintenance d'un tel environnement pour le développement d'applications réelles nous semble hors de portée. Rien ne nous interdit, en revanche, de retenir pour notre travail, l'idée architecturale qui consiste à placer le contrôle de l'exécution et des communications dans des modules séparés des descriptions proprement applicatives.

L'analyse du type de besoin dans notre activité de développement de systèmes TDAQ nous conduit donc naturellement à adopter *a priori* une approche de type « canevas ». Nous allons cependant voir que cette approche se distinguera des approches canevas classiques par l'adjonction d'une composante méthodologique.

12. Les développements de systèmes TDAQ menés dans tous les laboratoires du monde impliqués dans les grandes expérience de physique sont analogues à ce qui se fait au DAPNIA. En effet, il y peu de laboratoires dont la taille et les capacités en moyens et compétences soient très supérieures à celles du DAPNIA.

III.C.4 La généricité a priori *: une hypothèse irréaliste*

Que signifie pour nous « adopter une approche canevas » ? C'est se donner comme objectif final (au-delà même de cette thèse) de doter notre organisation (ici le DAPNIA/SEDI) d'un outil contenant des composants agencés selon une (ou des) architecture(s) générique(s) permettant de construire, par extension, configuration et assemblage, une application TDAQ. Il est clair que nous ne cherchons pas à modéliser tel ou tel système TDAQ particulier puisque nous nous proposons de saisir et de modéliser les structures communes à ces systèmes. Dans le même temps, au-delà de l'existence de structures communes à un niveau très abstrait (Cf. II.A), la variété extrême des systèmes TDAQ semble être un obstacle de taille à l'élaboration d'une véritable « application générique » permettant d'instancier, par quelques simples extensions et configurations, des applications réelles. En effet, si nous nous contentons de fabriquer une architecture « générique » par une simple analyse abstraite du domaine, nous serons finalement conduits à vouloir faire entrer de force l'architecture de nos applications futures dans le moule de notre architecture soi-disant générique. D'un point de vue technique une telle entreprise est probablement vouée à l'échec ; d'un point de vue humain, elle est irréaliste car l'expérience prouve que les cadres de travail qui ne sont pas profondément et organiquement adaptés aux exigences réelles d'une activité d'ingénierie ne parviennent jamais à s'implanter durablement au sein des équipes.

Ce qui va réellement conditionner la pertinence de notre canevas TDAQ sera la méthode avec laquelle nous allons le développer. Nous nous devons d'adopter une attitude *pragmatique* qui consistera à mettre en place une *méthodologie* de développement de systèmes TDAQ qui favorise *naturellement* l'élaboration de composants et d'architectures *réels* dans un souci de généricité. En d'autres termes, nous ne voulons pas fabriquer des composants *a priori* génériques : nous voulons fabriquer des composants réels pour des applications réelles, mais selon une ligne méthodologique qui favorise l'émergence de la qualité de généricité. En pratique, les modules génériques de notre canevas seront donc développés *à l'occasion* de telle ou telle application particulière mais leur conception devra bien entendu s'affranchir des spécificités de l'application en question.

III.C.5 La notion de « canevas méthodologique »

Notre objectif ne peut donc se limiter à repérer quelques structures, très abstrai-

tes, communes au systèmes TDAQ. Il s'agira de déceler non seulement les structures mais les mécanismes, contraintes et problématiques récurrents à l'œuvre dans le processus de développement de ces systèmes. L'objectif de ce travail est bien d'établir *une méthodologie et un cadre conceptuel* permettant de *baliser* le travail du concepteur de systèmes TDAQ et d'*encadrer* son activité afin de produire d'une part, des systèmes présentant une qualité interne améliorée et d'autre part des composants et architectures *réutilisables* qui pourront, à terme, s'assembler en ce qu'on pourra réellement appeler un « canevas TDAQ ». Comment élaborer ce cadre conceptuel ? En mettant à jour, au sein du travail de conception/réalisation d'un système TDAQ, la part qui correspond en réalité à l'application mécanique de principes de bonne programmation. L'explicitation de cette part « mécanique » du travail de conception/réalisation permettra, à terme, d'en envisager l'*automatisation* à l'aide d'outils informatiques *ad hoc*.

Il s'agit donc de débusquer – et, dans une certaine mesure, d'inventer – d'une part, les *motifs récurrents*[*] communs aux systèmes TDAQ eux-mêmes et d'autre part, les *motifs récurrents* au sein de *l'activité de développement* de ces systèmes, en ne retenant que ceux qui maximisent la généricité. L'intégration de ces deux classes de motifs récurrents dans un même cadre de développement aboutira à ce que nous appelons un « canevas méthodologique », c'est-à-dire un processus de développement formalisé permettant, d'une part, de baliser le travail des développeurs et d'encourager la production de modules réutilisables et, d'autre part, d'accumuler ces modules réutilisables au sein d'une architecture de développement générale convergeant au fil des développements vers un canevas proprement dit. Ce canevas sera, nous l'avons vu au III.B.5.3, essentiellement de type « boîte blanche », mais cela n'empêchera pas l'accumulation de composants réutilisables et de motifs récurrents programmés qui prendront place dans les menus « boîte noire » de solutions du canevas.

Remarquons enfin que la vocation du volet méthodologique est d'encadrer l'activité de modélisation des équipes et constitue donc *a priori* une *contrainte*. Si l'on peut, dans une certaine mesure, convaincre les personnes de la nécessité d'apprendre et d'appliquer ces contraintes (essentiellement conceptuelles) « à la main » par la formation et la communication, le meilleur résultat ne peut être obtenu que si l'on aboutit à terme à une *automatisation* de l'application de ces contraintes, c'est-à-dire à la mise en place d'un processus de développement

comprenant des phases de *transformation de modèle* automatisées. Aussi, pour être réaliste, les consignes de notre processus de développement devront-elles être définies de façon suffisamment précises (voire formelles) pour être automatisables.

IV

ARCHITECTURE ET CONTENU DU CANEVAS

Cette section constitue le cœur de ce travail de thèse. Elle est donc essentiellement consacrée à la justification et la présentation détaillée des principes, concepts et procédures qui constituent notre canevas méthodologique de développement de systèmes TDAQ. Dans la sous-section IV.A nous commençons par présenter les problématiques de distribution que nous devrons spécifiquement traiter dans notre canevas, vues sous l'angle des systèmes TDAQ. La sous-section IV.B sera consacrée à la spécification du cycle de développement dans lequel nous inscrivons notre canevas. Nous avons en effet constaté qu'il serait difficile, voire impossible, de développer un processus de *modélisation* de systèmes TDAQ sans le relier à un processus de *développement* plus général se traduisant par un cycle comprenant des étapes identifiées. La sous-section IV.C sera consacrée à la description détaillée de notre processus de modélisation proprement dit, sous une forme semi-formelle. Nous verrons enfin en IV.D comment ces processus de développement et modélisation de notre canevas méthodologique sont suceptibles de produire à terme un canevas classique de développement de systèmes TDAQ.

IV.A LES PROBLÉMATIQUES DE DISTRIBUTION

Pourquoi sommes-nous confrontés à une problématique de distribution dans les systèmes TDAQ ? Ainsi que nous l'avons vu au II.C.1, la tendance historique des systèmes TDAQ est à l'accroissement du nombre de nœuds de traitement parallèles dans les sous-systèmes qui les composent. Cette parallélisation croissante est due à des facteurs qui relèvent aussi bien de besoins en performances que de besoins de précision dans la mesure physique (Cf. II.A.1). Dans le même temps, un détecteur de physique est dévolu à *une seule* application qui nécessite l'usage simultané et coordonné de l'ensemble des sous-systèmes qui le composent pour pouvoir parvenir au résultat recherché. La problématique de distribution est donc à la fois un problème de *répartition* et de *regroupement* des tâches à accomplir. Nous avons ainsi besoin d'outils intellectuels et matériels pour pouvoir résoudre cette tension entre les nécessités contradictoires de répartir et de regrouper en parvenant à des solutions concrètes, réalisables et de qualité satisfaisante. A cette problématique *statique* de recherche d'une solution optimale s'ajoute une problématique *dynamique* qui est celle de pouvoir parvenir concrètement à la solution depuis les premières idées générales de départ jusqu'au déploiement effectif d'un

système complexe. En d'autres termes, il ne suffit pas de *trouver* une solution, encore faut-il trouver le chemin concret qui mène à la réalisation effective de cette solution. On peut ainsi affirmer que la recherche du processus de mise en œuvre d'une solution fait partie de la recherche de solution elle-même. Nous nous proposons donc dans le travail qui est présenté dans cette section IV d'établir un cadre conceptuel adapté au développement des systèmes TDAQ qui allie la recherche d'une solution au déploiement effectif de cette solution.

IV.A.1 Parallélisme intrinsèque et parallélisme de performance

IV.A.1.1 Parallélisme intrinsèque

Nous avons déjà évoqué au II.A.3 ainsi qu'au II.C.1 le caractère naturellement distribué des systèmes TDAQ. Ces systèmes sont en effet associés à des détecteurs qui sont répartis dans l'espace — parfois sur de très grandes distances. Les quelques exemples présentés en II.C.1 évoquent des dizaines de milliers (voire des millions) de canaux de détection qui doivent être numérisées et dont les données doivent être mise en forme, traitées puis stockées (voir, par exemple [B-23]). Schématiquement, nous pouvons dire qu'un détecteur de physique est composé d'un grand nombre de « nœuds de détection » géographiquement répartis. Cette répartition est imposée par les principes scientifiques qui fondent l'expérience. L'expérience ANTARES, par exemple, est fondée sur la détection de la lumière émise par des particules chargées de haute énergie dans l'eau ; cette lumière produit un signal dans les nœuds de détection, dont l'analyse et l'étiquetage spatio-temporel précis permettront de reconstruire l'énergie et la trajectoire de la particule. Or, pour que la mesure de la trajectoire, par exemple, soit suffisamment précise, il est nécessaire de disposer d'un bras de levier important, ce qui impose une répartition des nœuds de détection sur des distances importantes [B-24]. Une telle répartition est donc imposée par la physique et fait, pour ainsi dire, partie des caractéristiques intrinsèques de l'expérience. Elle donne lieu à un parallélisme du système TDAQ associé à la nature physique du détecteur. Pour cette raison, nous désignerons ce type de parallélisme (ou distribution) par l'expression « parallélisme intrinsèque » (ou « distribution intrinsèque »).

IV.A.1.2 Parallélisme de performance

Nous avons vu au II.C.1que les flots de données dans les expériences de

physique sont considérables et ont tendance à s'accroître fortement (Cf. fig. II-4). Aussi, la puissance de calcul exigée par le traitement en ligne de ces données ne peut-elle en général être fournie par une seule machine. C'est pourquoi, les systèmes TDAQ sont également caractérisés par ce qu'on peut appeler un « parallélisme de performance », c'est-à-dire une distribution du système qui découle de la nécessité de multiplier la puissance de calcul par l'ajout de processeurs de traitement supplémentaires. Les fermes de calcul en sont un exemple très fréquent : la partie à terre du système d'acquisition/déclenchement de l'expérience ANTARES est ainsi basée sur une ferme de calcul d'une centaine de stations de travail. De même, les études de performance du sous-système de déclenchement de niveau 2 d'ATLAS prédisent la nécessité de mettre en place une ferme de quelques centaines de processeurs [C-4].

IV.A.1.3 Impact sur la conception

IV.A.1.3.a Contraintes de robustesse et évolutivité

Le parallélisme intrinsèque découle de la distribution des nœuds de traitement situés près de l'électronique de détection. Ces modules contiennent de l'électronique spécifique et sont souvent enfouis et difficiles à atteindre. Par conséquent, s'ils sont donc susceptibles de connaître de fréquentes évolutions *durant la phase de conception*, ils n'évoluent presque plus une fois déployés. L'architecture de la distribution intrinsèque dans les systèmes TDAQ est par conséquent relativement statique durant le temps de vie du détecteur.

De plus, en raison de leur accessibilité physique réduite (voire nulle), les pannes dans les nœuds de traitement associés à la distribution intrinsèque ont un risque plus élevé de revêtir un caractère définitif. L'exigence de robustesse sur la conception de la partie intrinsèquement distribuée de l'expérience sera par conséquent plus grande.

Le parallélisme de performance, en revanche, apparaît le plus souvent sous la forme de fermes de processeurs accessibles et donc aisément réparables et améliorables. La partie d'un système TDAQ correspondant à ce type de parallélisme est donc susceptible de connaître un rythme d'évolution rapide, notamment pour pouvoir profiter au mieux des progrès de l'industrie électronique et informatique pendant toute la durée vie de l'expérience. De plus, son caractère extensible[1] le rend peu sensible aux pannes au sens où l'arrêt d'un processeur dégrade certes les

performances mais est peu susceptible de causer une panne générale. Bien entendu, des modules sensibles tels certains serveurs ou commutateurs peuvent entraîner de telles pannes générales, mais là encore, leur accessibilité les rend facilement remplaçables.

Ces différences qualitatives en termes de robustesse et d'évolutivité entre les parties affectées par le parallélisme intrinsèque et le parallélisme de performance impliquent que notre canevas de développement doit permettre de faire la distinction, et envisager des processus de conception différents pour chacune.

IV.A.1.3.b Modélisations différentes

Tout modèle représentant un ensemble de modules intrinsèquement parallèles découle directement de la spécification de l'application. Chaque module y remplit une ou plusieurs *fonctions* définies par la nature de l'application. Par exemple, dans l'expérience ANTARES, chaque module de détection est associé à une *position géographique* de détection, sachant que la nature même de l'expérience est fondée en partie sur la reconstruction de trajectoires de particules *à partir de points de détection* répartis dans un volume [A-3] [B-2]. La modélisation de l'application devra donc exprimer explicitement cette multiplicité des modules de détection car elle en est une partie intégrante.

En revanche, dans le cas du parallélisme de performance, la multiplicité n'est pas *en soi* une nécessité de l'application, puisqu'elle ne résulte que de la recherche de performance. Un module seul n'étant, dans la pratique, pas assez performant pour les besoins de l'expérience, on augmente la performance globale en rajoutant des modules supplémentaires[2]. Une modélisation classique du système va bien entendu exhiber plusieurs modules en parallèle mais rien ne va spécifiquement indiquer qu'il s'agit là de parallélisme de performance.

Considérons un exemple simplifié très représentatif des systèmes TDAQ en général. Imaginons que notre système soit constitué de 1000 nœuds de détection, que chaque événement doive être complètement assemblé* (Cf. II.A.3) avant de faire l'objet d'un traitement dans 100 processeurs du sous-système de déclenche-

1. *Nous qualifions d'« extensible » tout système susceptible de connaître une amélioration quantitative de ses performances par simple ajout de nouveaux modules. Le terme correspondant en anglais est « scalable ».*
2. *Le terme de « performance » peut ici désigner autre chose que de la puissance de calcul ; il peut également s'agir de capacité de stockage, de bande passante, etc.*

ment (Cf. figure IV-1). Il est clair que la multiplicité des processeurs de traitement du sous-système de déclenchement correspond à un parallélisme de performance, alors que les 1000 nœuds de détection correspondent à une distribution intrinsèque du détecteur.

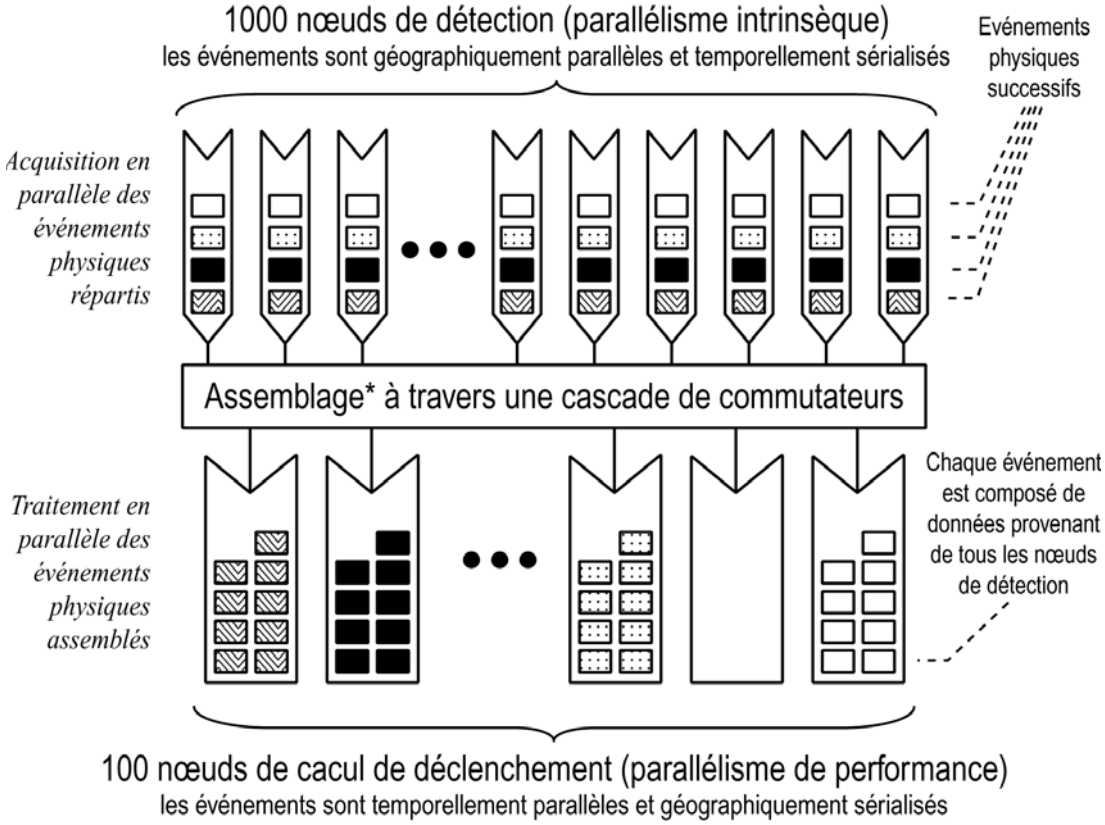


Fig. IV-1 : Parallélismes dans une topologie typique d'un système TDAQ

La « multiplicité » due à la recherche de performance est ici *de nature différente* par rapport à la multiplicité applicative intrinsèque. Nous considérons que la modélisation du système doit expliciter cette différence car les problématiques de conception/réalisation diffèrent dans chacun des cas. Tentons de modéliser sommairement les objets « nœud de détection » et « module de traitement ». Un modèle structurel simplifié en UML serait le diagramme de la figure IV-2. Nous y voyons en particulier que l'association entre `Nœud_Detection` et `Nœud_Declenchement` comporte les deux cardinalités 1000 et 1..100 qui traduisent les deux parallélismes de notre système. Bien que nous ayons choisi de spécifier un intervalle (1..100) au lieu d'une valeur fixe pour la cardinalité qui concerne le parallélisme de performance afin de suggérer l'idée d'un ensemble extensible* (*scalable*) de processeurs, rien, sur ce diagramme, ne permet réelle-

ment de savoir que la raison pour laquelle le système comporte plusieurs objets `Nœud_Declenchement` est un besoin de performance alors que la multiplicité des objets `Nœud_Detection` est nécessaire du fait de la fonctionnalité du système. Nous verrons comment utiliser les mécanismes d'extension UML pour introduire explicitement cette spécification.

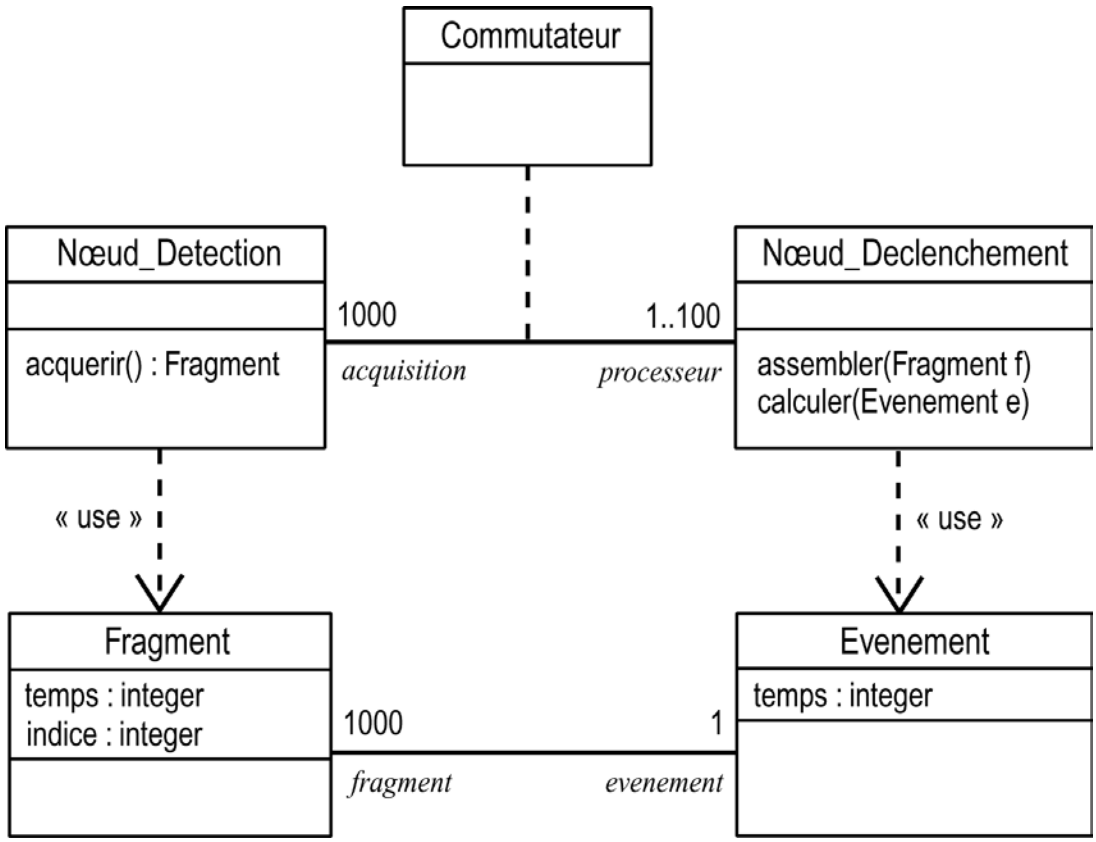


Fig. IV-2 : Diagramme de classes simplifié du système TDAQ

IV.A.2 Plateformes matérielles et logicielles

Ainsi qu'évoqué en II.A.1 et en II.A.5, les expériences de physique fondamentale sont presque toujours basées sur des détecteurs particuliers qui nécessitent donc le plus souvent une électronique de lecture spécifique. Derrière un canal de détection physique – par exemple un photomultiplicateur ou une sortie de CCD – une électronique spécifique doit mettre le signal en forme, le numériser puis transmettre les données résultantes au système de traitement/acquisition qui, à son tour, effectue certains traitements en temps réel sur les données produites et en gère le flot (Cf. fig. IV-3). Dans cette chaîne de traitement du signal, il est souhaitable que l'électronique spécifique cède le pas au logiciel le plus en amont possible, et ce, principalement pour 2 raisons :

- Les développements électroniques sont coûteux en efforts, il faut donc les réduire au maximum (Cf. II.A.5).
- Les plateformes logicielles sont plus flexibles et donc plus adaptables aux évolutions des algorithmes de traitement des données (Cf. II.A.7).

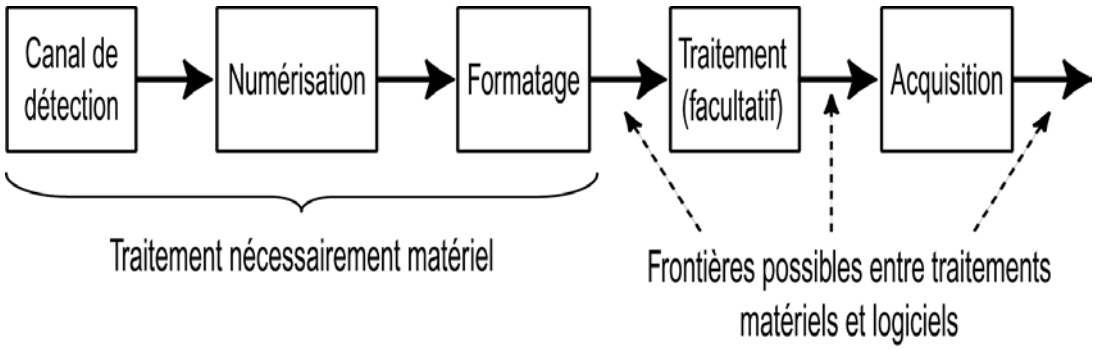


Fig. IV-3 : Frontière entre matériel et logiciel dans la chaîne de détection

Les cycles de développement et de fonctionnement d'un grand détecteur de physique étant très longs par rapport au rythme de progression des industries électronique et informatique, il demeure très hasardeux de fixer la frontière entre matériel et logiciel trop à l'avance. Pour bénéficier des progrès rapides de la technique (en particulier *pendant* le cycle de développement) et minimiser notre effort de développement, il convient que notre canevas TDAQ soit relativement *flexible* sur cette question. En effet, la méthodologie de développement devrait autoriser l'implémentation des différents modules de traitement sur différentes plateformes (matérielles ou logicielles). Typiquement, un module de calcul qui, pour des raisons de performances, doit être implémenté sous forme de circuit électronique spécifique, peut finalement l'être sous une forme logicielle grâce aux progrès rapides des processeurs.

Le canevas doit donc autoriser, dans une certaine mesure, un processus de conception qui tolère, et même prévoit, les re-partitionnements entre matériel et logiciel. Le résultat de la conception du système doit pouvoir être *redistribué* sur une (ou des) configuration(s) matérielle(s) particulière(s). Ainsi, le partitionnement matériel/logiciel peut être vu comme un problème de distribution : si sur 2 modules A et B, A est vu comme un composant matériel écrit en VHDL et B comme du logiciel écrit en Java, le concepteur devra spécifier les interfaces et le protocole de communication entre les deux (chaque côté étant bien entendu implémenté dans le langage concerné). La problématique de définition de la

frontière entre matériel et logiciel est ramenée à une problématique de distribution.

IV.B Le cycle de développement proposé

Nous avons vu au II.B un aperçu du cycle de développement implicite suivi par l'activité d'ingénierie des systèmes TDAQ. Ce cycle possède quelques traits communs avec les cycles « en spirale » de la littérature [C-28], en raison de son caractère incrémental et son orientation « essais-erreurs ». Il rappelle également les techniques de « prototypage évolutif » puisqu'une grande partie du travail consiste à concevoir et mettre en œuvre des systèmes prototypes que l'on fait ensuite croître progressivement vers le système complet[3]. Nous pensons que la complexité et la longévité des développement de systèmes TDAQ nécessite une telle orientation de principe. Ainsi qu'évoqué au II.B.2, le défaut principal du processus de développement TDAQ aujourd'hui réside dans son caractère essentiellement centré sur l'architecture matérielle, le logiciel n'y ayant que le statut de code rattaché aux modules matériels. Nous nous proposons donc de reprendre avec quelques modifications les principes de bases de ce processus, en les formalisant davantage et, surtout, en adoptant une approche intégrée matériel/logiciel.

IV.B.1 Vue d'ensemble

La figure IV-4 donne une représentation possible de notre nouveau cycle de développement. A l'instar du diagramme d'un cycle en spirale, le temps s'écoule du centre vers la périphérie. Nous avons décomposé ce cycle en cinq classes d'activités (encadrées par un rectangle arrondi) :

- Description fonctionnelle
- Elaboration d'architecture
- Etudes d'implémentation, partitionnement matériel/logiciel
- Développement de prototypes
- Tests et mesures

3. *Dans cette expression, le terme de « prototypage » désigne habituellement le prototypage logiciel, c'est-à-dire l'élaboration d'un squelette de code où la détermination de l'architecture et des interfaces prime sur le contenu proprement algorithmique. Ici, le terme est également entendu au sens classique de prototype matériel. Dans les deux cas, le « prototypage évolutif » consiste à faire évoluer les prototype vers le système final proprement dit.*

L'épaisseur des zones hachurées indique l'importance de l'activité au fur et à mesure de l'avancement du cycle. Sont également présentes les études physiques par simulation Monte-Carlo et les campagnes de mesures physiques (Cf. fig. II-3) : les résultats de ces études sont les *inputs* de la physique dans ce cycle de développement. Ils permettent d'estimer les besoins en performances du système et d'élaborer ainsi une architecture qui soit susceptible d'y répondre. Les études Monte-Carlo interviennent également plus en aval pour évaluer l'impact des solutions envisagées sur la qualité des données physiques, ce dont nous avons vu l'importance au II.A.4.

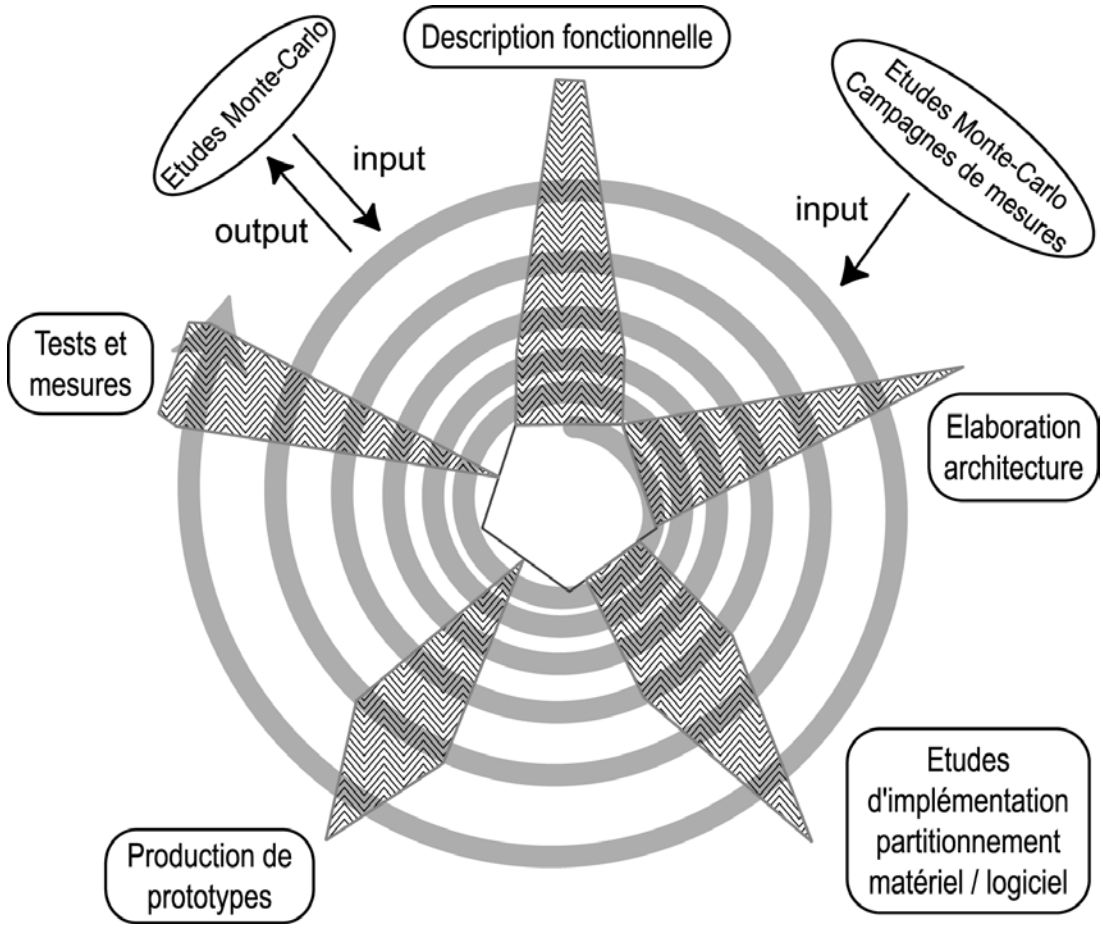


Fig. IV-4 : Cycle de développement TDAQ modifié

IV.B.2 Description fonctionnelle

La description du système que nous qualifions de « fonctionnelle » correspond à l'expression de l'application que nous voulons développer sans considération aucune pour les questions d'implémentation. Cette description est en fait la traduction libre de ce que nous voulons mesurer du point de vue de la physique en un

instrument abstrait qui ne s'embarrasse pas, au départ, des limitations de performances liées à l'acquisition et aux traitements en ligne. D'après les études physiques, nous savons que pour obtenir le résultat scientifique escompté nous devons être capables de mesurer, à partir des technologies de détection disponibles, telle quantité de trajectoires, d'énergies et autres grandeurs physiques avec telle précision. Cette réflexion doit nous permettre de spécifier, de plus en plus précisément au cours du cycle de développement, les *fonctions* que doit remplir le système TDAQ envisagé.

Concrètement, la description fonctionnelle du système va donc comprendre la spécification des *objets* liés à l'accomplissement des fonctions ainsi que leur *nombre*. Par exemple, les énoncés suivants, concernant l'expérience ANTARES, sont typiques d'une description fonctionnelle[4] :

- L'instrument complet comprendra 10 colonnes de détection réparties sur une surface de 0,1 km^2. Chaque colonne comprendra 30 modules de détection optiques (MO) sur une hauteur totale de 300 m.
- Chaque MO comprendra un triplet de photomultiplicateurs* (PM) ; chaque PM détectera un taux d'événements primaires d'environ 60 kHz.
- Les signaux devront être numérisés, et étiquetés temporellement grâce à un unique signal d'horloge de 20 MHz réparti sur tous les modules de détection.
- Les données résultant de la numérisation devrons être formatées puis, soit filtrées (rejetées), soit stockées sous forme de groupes temporellement corrélés[5]. L'algorithme de déclenchement principal mis en œuvre fait intervenir les signaux temporellement corrélés de l'ensemble du détecteur.

A ces spécifications, nous pouvons ajouter les contraintes connues *a priori* :

- Le maintien mécanique de chaque colonne s'effectue à l'aide d'un câble électromécanique qui constitue également la gaine par laquelle passent toutes les liaisons (optiques ou électriques) entre modules optiques et stations à terre. La nécessité de pouvoir enrouler chaque colonne à bord d'un bateau

4. Ces énoncés ne correspondent pas tous exactement à la réalité car nous en avons volontairement simplifié certains dans un souci d'intelligibilité didactique. En particulier, les questions de la numérisation des signaux et du déclenchement sont plus complexes.

5. C'est-à-dire appartenant à une même fenêtre temporelle.

de déploiement impose des limites strictes sur la rigidité et donc le diamètre du câble électromécanique : le nombre de liaisons entre modules optiques et stations à terre devra donc être minimisé.

- L'évacuation de la chaleur dans chaque étage de détection ne peut se faire que par simple convection à travers les parois en alliage de titane du conteneur de l'étage. D'autre part, la puissance électrique disponible est également limitée du fait des restrictions sur le dimensionnement des câbles. La consommation électrique devra par conséquent être minimisée.

Ces énoncés permettent d'élaborer les éléments d'un premier modèle fonctionnel sous la forme de diagrammes de séquences. Par exemple, la figure IV-5 représente la séquence relative à la numérisation d'un signal de photomultiplicateur. La figure suivante (Fig. IV-6) représente le déclenchement sur les données d'un module optique. Tous les objets introduits découlent directement de la description fonctionnelle ci-dessus, en tenant éventuellement compte des contraintes *a priori*.

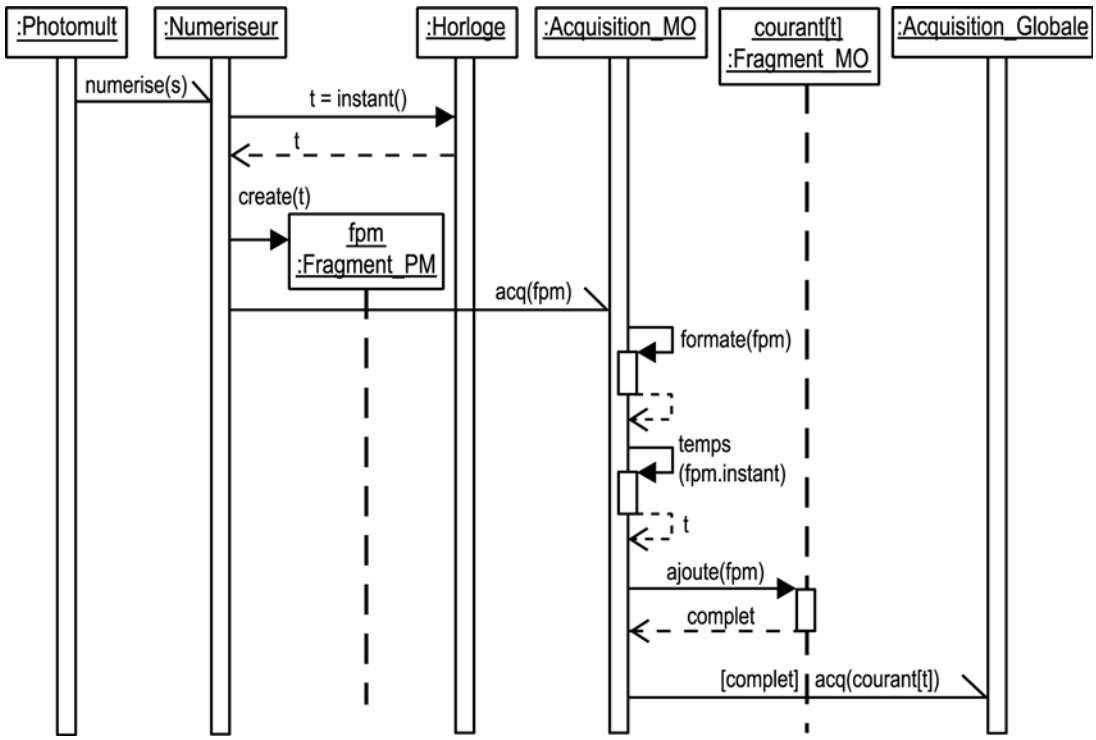


Fig. IV-5 : Séquence de numérisation / acquisition d'un signal de photomultiplicateur (PM)

Comment en arrive-t-on à ces séquences à partir des spécifications et contraintes énoncées ? L'exemple de raisonnement suivant illustre ce processus de conception : les limitations sur le diamètre de câble ainsi que les distances importantes entre modules optiques mènent naturellement à l'utilisation de liaisons

optiques. La nécessité d'en limiter le nombre impose également que l'on fasse pleinement usage de leurs capacités en bande passante. Il est donc exclu que chaque photomultiplicateur bénéficie de sa propre liaison jusqu'à la terre. On doit ainsi effectuer un assemblage (Cf. II.A.3) partiel des données de l'événement au niveau d'un module optique, assemblage pris en charge par les objets de la classe `Acquisition_MO` (fig. IV-5). A chaque fois qu'un tel assemblage partiel est effectué, un objet de classe `Fragment_MO` contenant les données temporellement corrélées d'un même module optique est transmis à l'objet d'assemblage global de classe `Acquisition_Globale`. Celui-ci assemble l'événement global de classe `Evenement` et le transmet à l'objet `:Declenchement` (fig. IV-6) qui renvoie sa réponse après calcul dans la variable `rejet`. Selon cette réponse, l'objet `:Acquisition_Globale` transmettra l'événement au système de stockage ou le détruira (fig. IV-6).

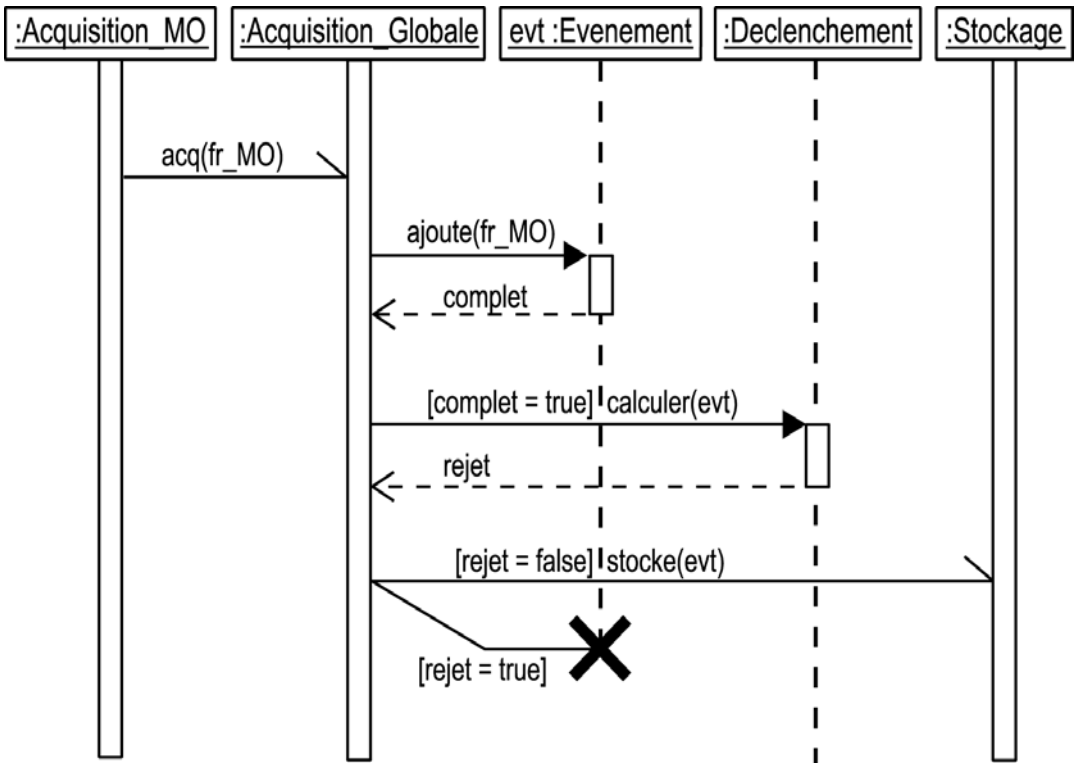


Fig. IV-6 : Séquence d'acquisition / déclenchement d'un événement

A ces modélisations, nous pouvons ajouter des grandeurs, elles-mêmes purement fonctionnelles, correspondant aux nécessités intrinsèques de l'expérience. Par exemple, nous avons spécifié que les photomultiplicateurs détecteraient des signaux à un taux de 60 kHz. Ce chiffre est intrinsèque aux caractéristiques du PM couplées au bruit naturel émis dans le milieu de l'expérience, à savoir, -2000 m

à 40 km au large de Toulon. Par ailleurs, la précision nécessaire à l'expérience fixera les paramètres de numérisation de ces signaux, tels la dynamique et le taux d'échantillonnage. On pourra donc en déduire une donnée fonctionnelle déterminante pour la conception du système TDAQ : le débit numérique moyen produit à la sortie des numériseurs (il est, pour ANTARES, de l'ordre de 7 Mb/s, cf. [C-5] [B-25]). Bien entendu, comme pour le modèle fonctionnel lui-même, ces données seront précisées et détaillées au cours du cycle de développement, notamment grâce aux résultats des campagnes de mesures et des simulations par Monte-Carlo. Il faudra par exemple déterminer les fluctuations du débit numérique autour de sa valeur moyenne, grandeur déterminante pour l'architecture et le dimensionnement de l'acquisition des modules optiques.

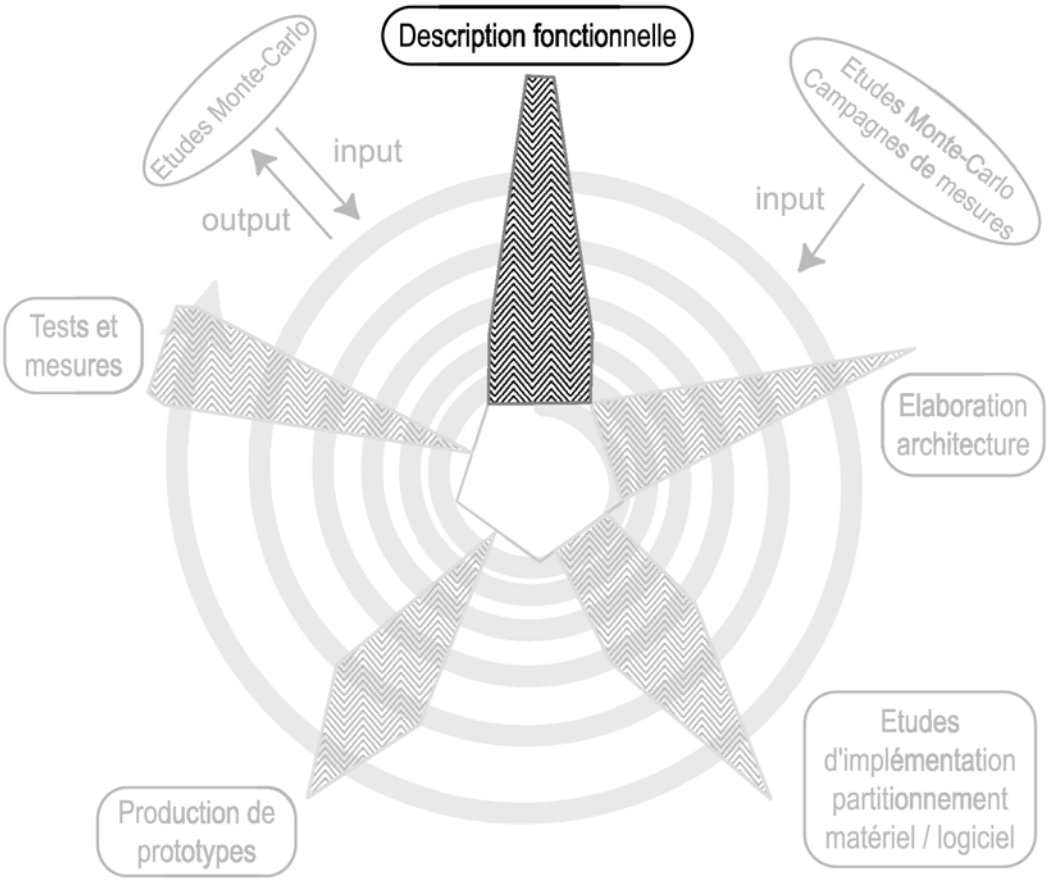


Fig. IV-7 : Phase fonctionnelle du cycle de développement TDAQ

La figure IV-7 nous montre également que dans la spirale du cycle de développement, la spécification fonctionnelle a bien entendu beaucoup d'importance en début et en milieu de cycle pour s'amenuiser progressivement jusqu'à la fin du cycle. En effet, plus l'activité de développement avance, plus l'élaboration fonctionnelle cède la place aux problématiques d'implémentation et aux réalisations

concrètes. Il demeure cependant presque toujours une activité résiduelle concernant la description fonctionnelle (l'extrémité du pétale de la description fonctionnelle sur la figure IV-4 n'est pas d'épaisseur nulle), puisque certains algorithmes de déclenchement/sélection de haut niveau continuent souvent à être modifiés et affinés, même après le démarrage de l'expérience.

IV.B.3 Elaboration d'architecture

Les spécifications fonctionnelle du système permettent d'en élaborer l'architecture de principe. La topologie intrinsèque du détecteur impose ses contraintes en matière d'architecture matérielle, notamment le découpage en modules de détection. L'apport des campagnes de mesures et des simulations par Monte-Carlo permet de compléter et de préciser les besoins en performances, ce qui va également orienter l'élaboration de l'architecture.

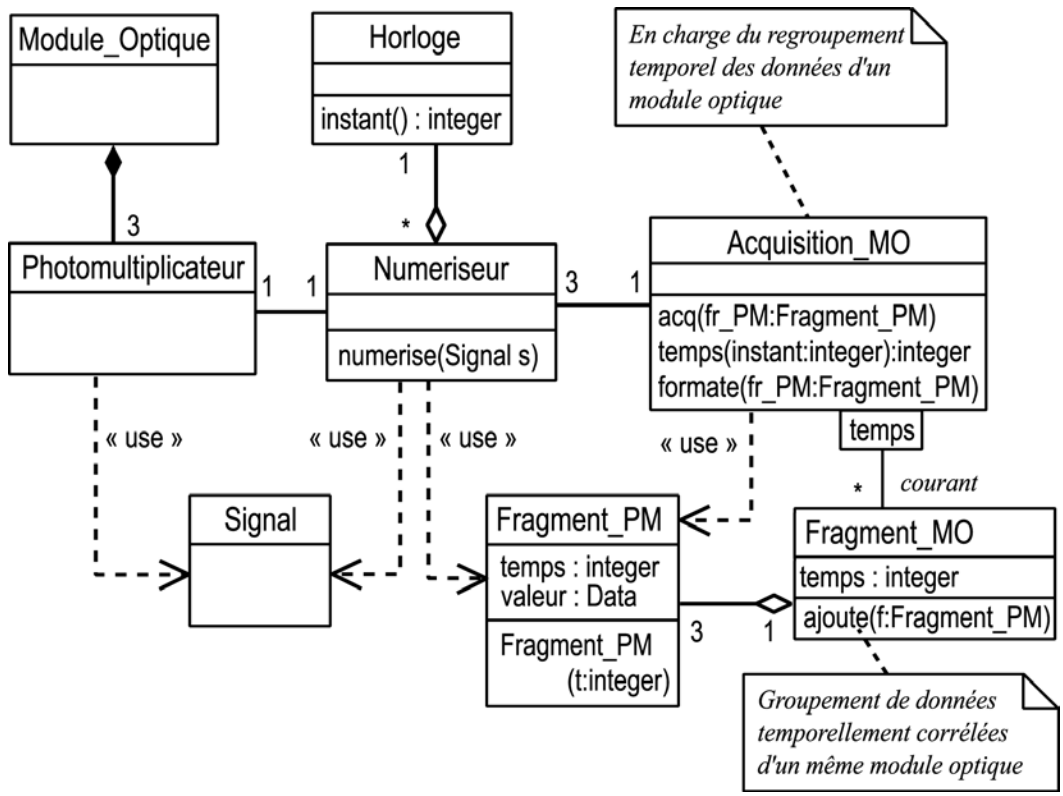


Fig. IV-8 : Classes des objets intervenant dans la numérisation du signal PM

Considérant l'exemple inspiré d'ANTARES présenté au IV.B.2, nous pouvons déduire directement une première description architecturale sous la forme des deux diagrammes de classes représentés sur les figures IV-8 et IV-9, respectivement reliés aux diagrammes de séquence IV-5 et IV-6. Sur ces diagrammes, l'ensemble des classes, leurs associations *ainsi que toutes les cardinalités* découlent directe-

ment des spécifications fonctionnelles. Nous avons par exemple besoin de 3 numériseurs pour 1 module optique, ce qui détermine les cardinalités de l'association entre `Acquisition_MO` et `Numeriseur`. De même, l'acquisition globale est effectuée par un seul objet de type `Acquisition_Globale` associé à l'ensemble des objets `Acquisition_MO`.

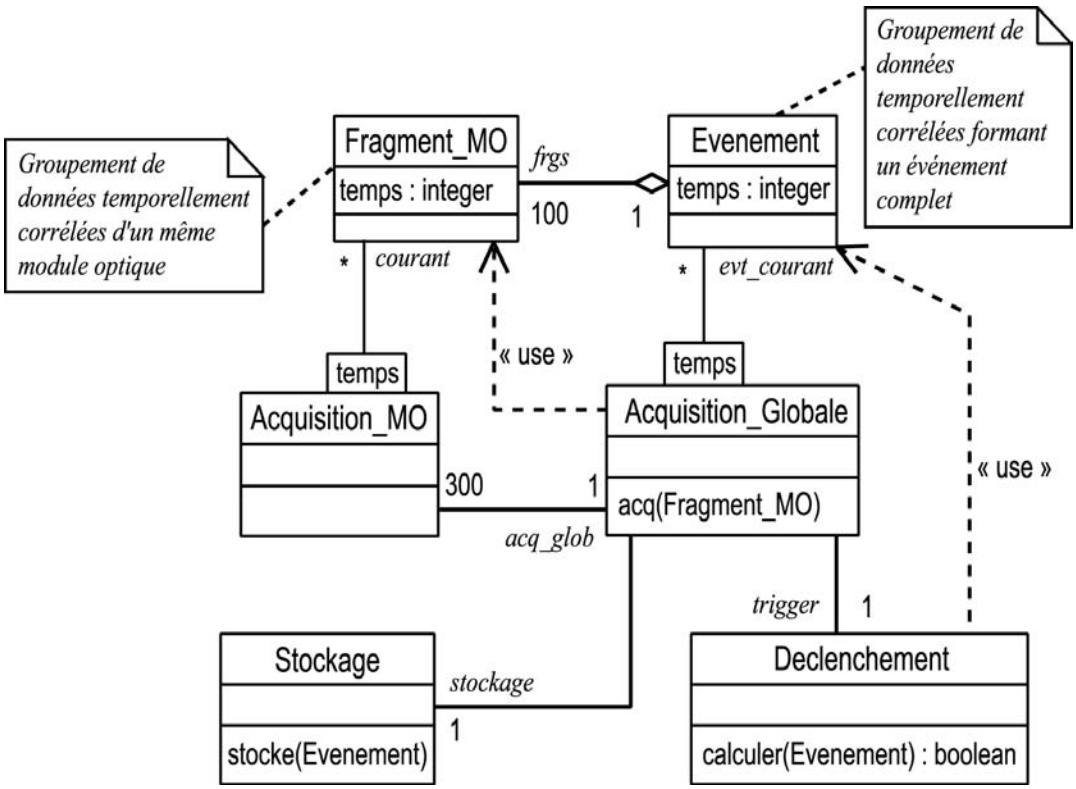


Fig. IV-9 : Classes des objets intervenants dans l'acquisition globale des données du MO

On conçoit cependant assez bien qu'après un premier cycle [Description fonctionnelle] → [Elaboration d'architecture] → [Etude d'implémentation] (les activités [Production de prototypes] et [Tests et mesures] étant inexistantes au début du cycle ; voir fig. IV-4), les contraintes de performances puissent, par exemple, imposer que l'objet `:Acquisition_Globale` soit dédoublé afin de pouvoir absorber le débit de données entrant. Ce dédoublement, c'est-à-dire une cardinalité de `2` (ou plus) au lieu de `1` affectée au rôle `acq_glob` dans l'association entre `Acquisition_MO` et `Acquisition_Globale` (fig. IV-9), correspondrait à un parallélisme de performance tel que défini au IV.A.1.2 et serait typiquement une conséquence de l'analyse des besoins en performances et d'une première étude d'implémentation.

L'activité d'élaboration de l'architecture consiste donc à déterminer la structuration du système étant données les fonctionnalités attendues. Cette activité ne

distingue pas *a priori* entre architectures matérielle et logicielle et doit se contenter uniquement d'exprimer quels sont les objets dont on a besoin et quels rapports ils entretiennent entre eux.

Il n'en demeure pas moins que le point de départ de la réflexion sur l'architecture système est donné par la structuration *physique* du détecteur, c'est-à-dire un point de vue essentiellement *matériel*. En effet, les instruments de détection primaires qui produisent le signal analogique sont bien entendu des composants matériels du système. Le logiciel ne peut commencer à exister qu'à partir du point où le signal est numérisé (Cf. fig. IV-3, p. IV-9) ou bien dans tout ce qui concerne le *contrôle* des instruments. Nous savons également que si, par exemple, nos modules de détection sont très éloignés les uns des autres, nous devrons les relier aux objets d'acquisition à travers des liens matériels de communication. Nous avons donc, au début du travail de conception, un squelette du « substrat matériel » du système.

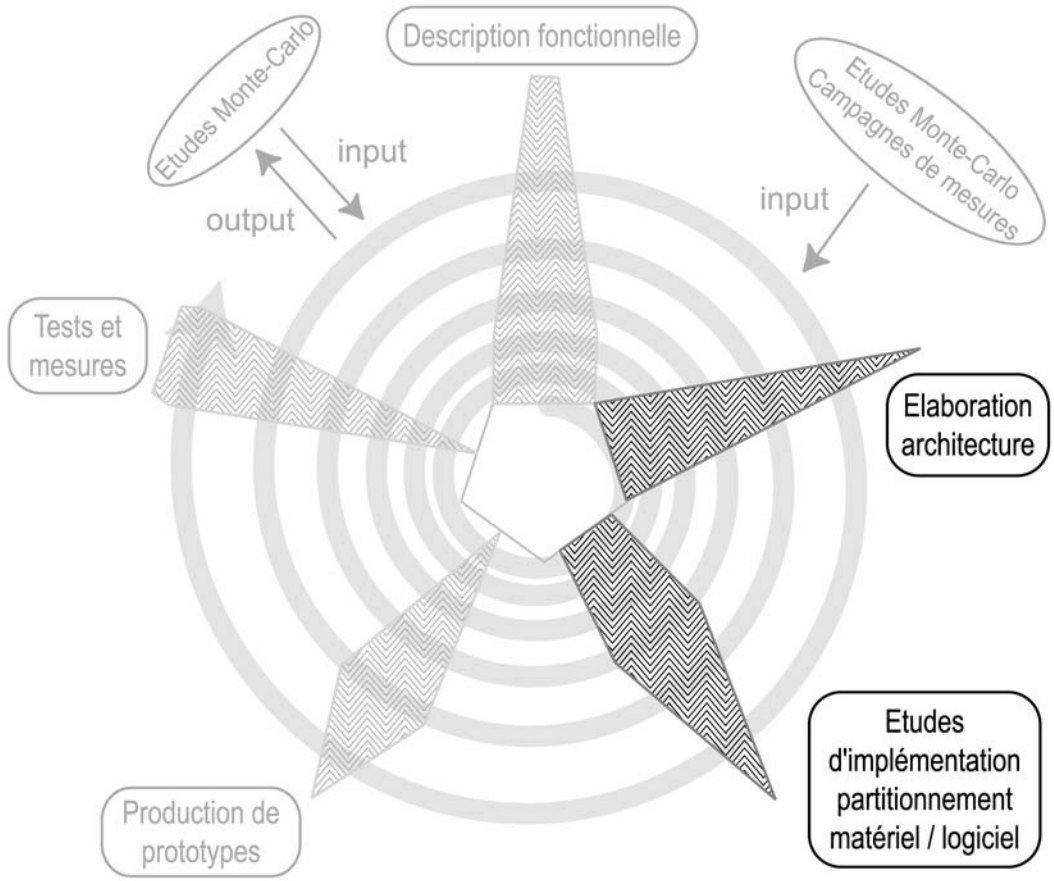


Fig. IV-10 : Phase d'élaboration d'architecture et d'implémentation/partitionnement du cycle de développement TDAQ

Ce qui n'est pas déterminé dès le départ quant à sa nature matérielle ou logiciel-

le, ce sont les objets qui mettront en œuvre d'une part, les fonctionnalités « fondamentales » d'acquisition et de traitement des signaux numérisés et d'autre part, les fonctionnalités « périphériques » de communication, de contrôle, d'ordonnancement, etc. Aussi, lorsque nous disons que l'élaboration de l'architecture ne distingue pas *a priori* entre matériel et logiciel, cela ne concerne bien entendu que ces objets situés à un niveau plus haut que le substrat matériel primaire du détecteur. D'autre part, l'élaboration de *l'architecture* du système sera bien évidemment influencée en profondeur par le partitionnement matériel/logiciel, mais seulement *a posteriori*. Par exemple, le passage à une cardinalité plurielle pour des raisons de performances peut être la conséquence d'une option d'implémentation matérielle tel le choix d'utiliser tel processeur peu puissant pour des raisons de consommation électrique.

Enfin, nous pouvons voir sur la figure IV-10 que l'épaisseur du pétale relatif à l'élaboration d'architecture décline progressivement vers zéro au fur et à mesure de l'avancement du cycle de développement. Il est en effet clair qu'aux stades avancés du projet, il n'est plus question de modifier profondément l'architecture du système : celle-ci se stabilise et cède le pas à sa mise en œuvre détaillée.

IV.B.4 Etudes d'implémentation, partitionnement matériel/logiciel

A partir des spécifications fonctionnelles du système, de ses besoins en performances et d'une architecture de principe, on peut envisager d'étudier l'implémentation du système. Cette activité ne comprend pas la fabrication de prototypes ni les mesures réelles que nous considérons comme des phases du cycle à part entière. Il s'agit, au cours de cette phase, d'effectuer un travail essentiellement intellectuel sur la mise en œuvre concrète du système à partir des contraintes quantitatives connues de l'application et de l'offre indutrielle. Comme indiqué sur la figure IV-10 par l'épaisseur du pétale associé, cette activité démarre tôt dans notre cycle de développement car il est nécessaire d'établir une faisabilité de principe afin d'orienter les réalisations qui permettront d'aboutir aux choix définitifs. On peut voir sur la figure que cette activité augmente dans un premier temps, au fur et à mesure que la description fonctionnelle avance, puis se réduit jusqu'à s'annuler à mesure que l'on passe aux réalisations concrètes et aux tests et mesures réels.

Nous pouvons subdiviser les études d'implémentations en les points suivants

(l'ordre est indifférent) :

- Effectuer un partitionnement matériel/logiciel du système
- Compléter l'architecture du substrat matériel
- Choisir les technologies matérielles de la partie matérielle
- Choisir les technologies matérielles de la partie logicielle
- Choisir les technologies logicielles de la partie logicielle

Les différents choix à effectuer au cours de cette activité doivent bien entendu se fonder sur la connaissance et l'étude du marché des technologies matérielles et logicielles utilisables. En fait, la veille technologique et le maintien à jour des connaissances techniques des ingénieurs sont la base indispensable à l'efficacité de l'activité « Etudes d'implémentation et partitionnement logiciel/matériel ». Cependant, par souci de ne pas trop nous disperser, nous n'approfondirons pas ce volet et nous nous en tiendrons à la spécification des mécanismes de choix au cours de cette activité.

IV.B.4.1 Effectuer un partitionnement matériel/logiciel du système

La problématique la plus fréquente rencontrée dans cette activité est la recherche de compromis entre la souplesse conférée par les solutions logicielles et les meilleures performances atteignables par les solutions matérielles. Afin d'analyser comment doit s'effectuer le traitement de la problématique, poursuivons sur notre exemple du système d'acquisition d'ANTARES.

Considérons à nouveau la figure IV-5, reproduite ci-après sur la figure IV-11. Il est clair que, de par leur fonction, le numériseur et l'horloge sont nécessairement des objets matériels (Cf. IV.A.2 et fig. IV-9). La question du partitionnement matériel / logiciel va donc se poser à partir de l'objet `:Acquisition_MO`.

En ce qui concerne le numériseur, les spécificités de l'expérience ANTARES ont conduit la collaboration à opter pour le développement de l'ARS, un ASIC* permettant d'effectuer une procédure complexe de numérisation fondée sur une discrimination entre différentes formes du signal [B-27]. Chaque module optique est doté, pour des raisons physiques, de 3 photomultiplicateurs, et chaque photomultiplicateur sera échantillonné par une paire d'ARS afin de réduire le temps mort de numérisation. Les spécifications de fonctionnement et d'entrées/sorties de l'ARS

vont donc contraindre la question du partitionnement. Nous savons par exemple que des paquets de données de formats très différents seront produits par l'ARS et que son débit maximum en sortie est de 20 Mb/s. Les campagnes de mesures indiquent également que les fluctuations des taux de signaux détectés seront suffisamment fortes pour que ce débit maximum soit régulièrement atteint, sachant que le débit moyen en sortie d'une paire d'ARS sera de l'ordre de 7 Mb/s [C-5]. Certaines fluctuations, dues à la bioluminescence[6], peuvent durer jusqu'à plusieurs secondes. Il apparaît donc clairement que le module d'acquisition d'un même étage de détection devra d'une part, être capable d'absorber des pics de débit de 120 Mb/s (20 Mb/s × 6 ARS) pendant plusieurs secondes et d'autre part, écouler l'ensemble des données vers les stations à terre à un débit supérieur à 21 Mb/s.

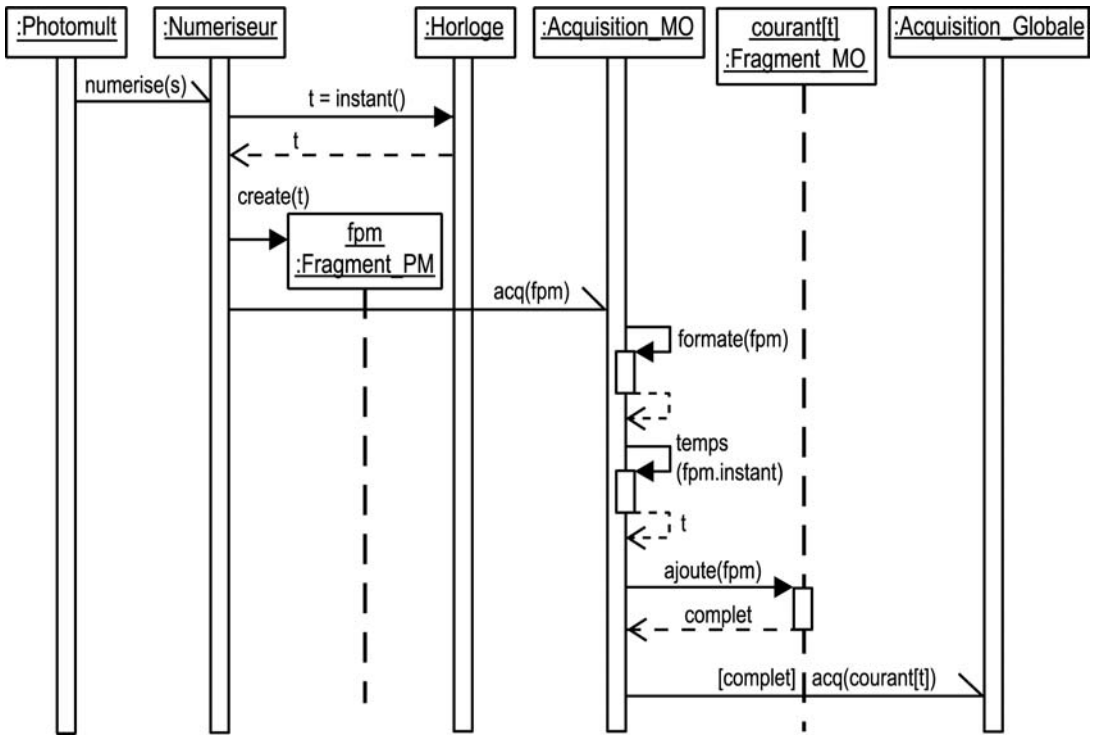


Fig. IV-11 : Séquence de numérisation / acquisition d'un signal de photomultiplicateur (PM)

Nous voyons ainsi que l'acquisition d'un module optique est en fait constituée de deux versants clairement identifiés : un versant « amont » qui requiert des performances élevées car il doit absorber des paquets de données complexes à des taux fortement fluctuants, et un versant « aval » dont la fonction est essentiellement de transmettre les données à un rythme régulier aux stations à terre. Cette analyse n'équivaut bien entendu pas au traitement du système réel, bien plus élaboré, mais elle représente bien le cheminement suivi. Elle nous mène ainsi naturellement vers

6. *Phénomène de production de lumière par les organismes vivants des abysses.*

une décomposition de l'objet `:Acquisition_MO` en deux objets `:Lecture_ARS` et `:Transmission` respectivement mis en œuvre sous forme matérielle et logicielle. En effet, les exigences de performance du versant « amont » imposent une solution matérielle alors qu'une solution logicielle semble envisageable pour le versant « aval ».

Pourquoi s'orienter vers une implémentation logicielle pour ce versant « aval » ? Ne pourrait-on intégrer l'ensemble de l'objet `:Acquisition_MO` en un seul module matériel ? Probablement non, car nous savons que le système final comportera des centaines de modules optiques et que la régulation des flux de données associés nécessitera probablement une mise au point algorithmique qui ne peut se faire qu'à l'aide d'un système possédant la souplesse du logiciel. De plus, comme souligné au II.A.5, les solutions logicielles sont *a priori* préférées lorsqu'elles sont possibles en raison de leur coût de développement et de maintenance inférieur. Dans notre exemple, une solution purement matérielle devrait mettre en œuvre non seulement les fonctionnalités fondamentales du système mais également toute l'électronique nécessaire à l'interfaçage avec le matériel industriel utilisé pour la communication avec les stations à terre.

Cette première analyse nous conduit donc à affiner notre modèle fonctionnel (fig. IV-12, IV-13, IV-14) et à en dresser un premier modèle d'implémentation (fig. IV-15).

La figure IV-12 représente ainsi l'évolution du diagramme de séquence de la figure IV-11suite à notre analyse d'implémentation. Dans le diagramme résultant, l'objet d'acquisition initial de type `Acquisition_MO` a été remplacé par deux objets de classes respectives `Lecture_ARS` et `Transmission`. Le premier aura la charge de la lecture « rapide » des numériseurs et la constitution des fragments de données temporellement corrélées de l'étage de détection, alors que le deuxième assurera la transmission de ces données aux stations à terre à débit quasi constant. Le diagramme de séquence « aval » de la figure IV-6 (Séquence d'acquisition / déclenchement) doit également être modifié en y remplaçant l'objet `:Acquisition_MO` par l'objet `:Transmission`.

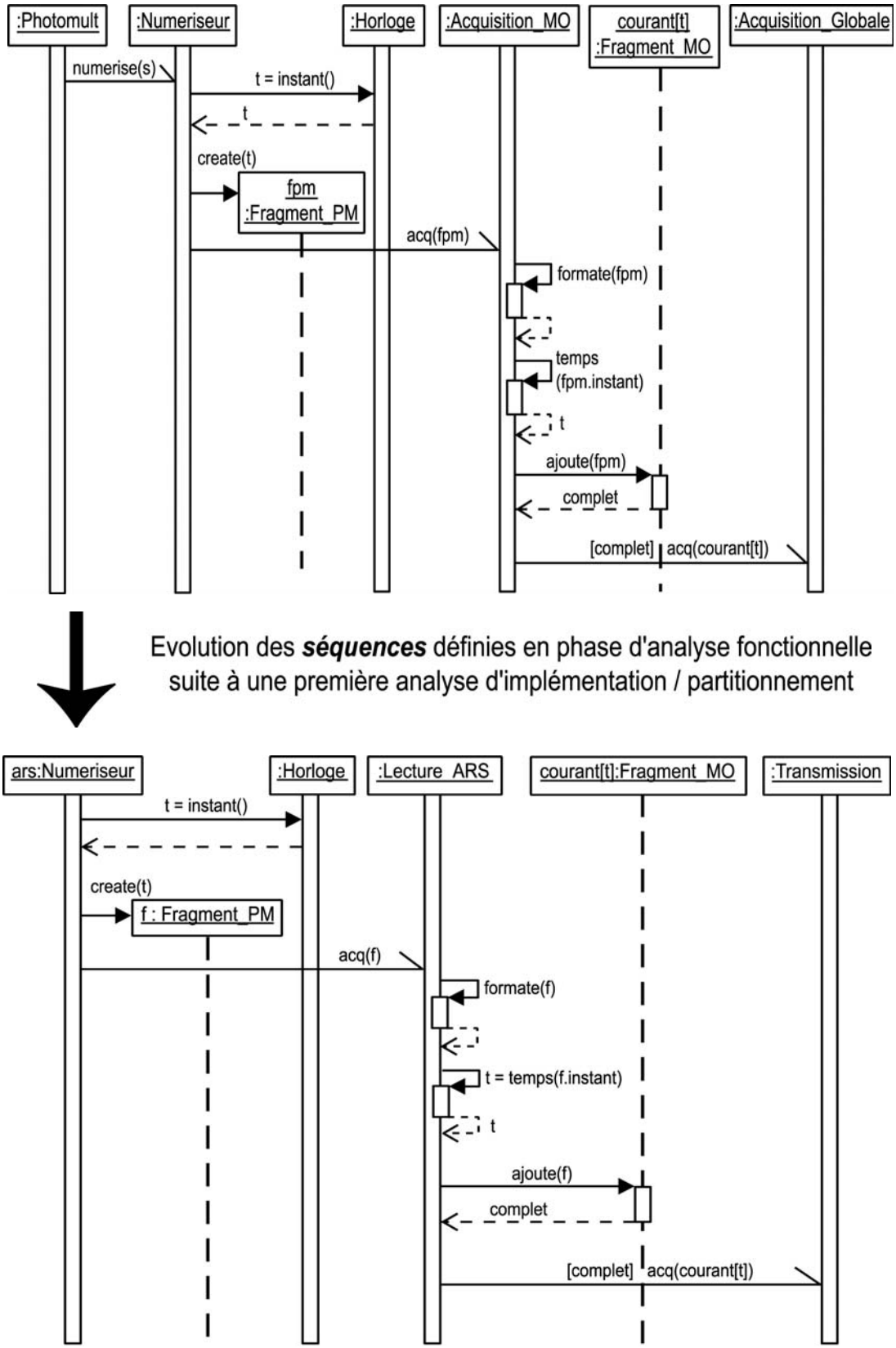


Fig. IV-12 : Evolution du modèle fonctionnel de numérisation / acquisition d'un signal de PM (l'interaction matérielle avec le PM a été omise afin d'alléger le diagramme)

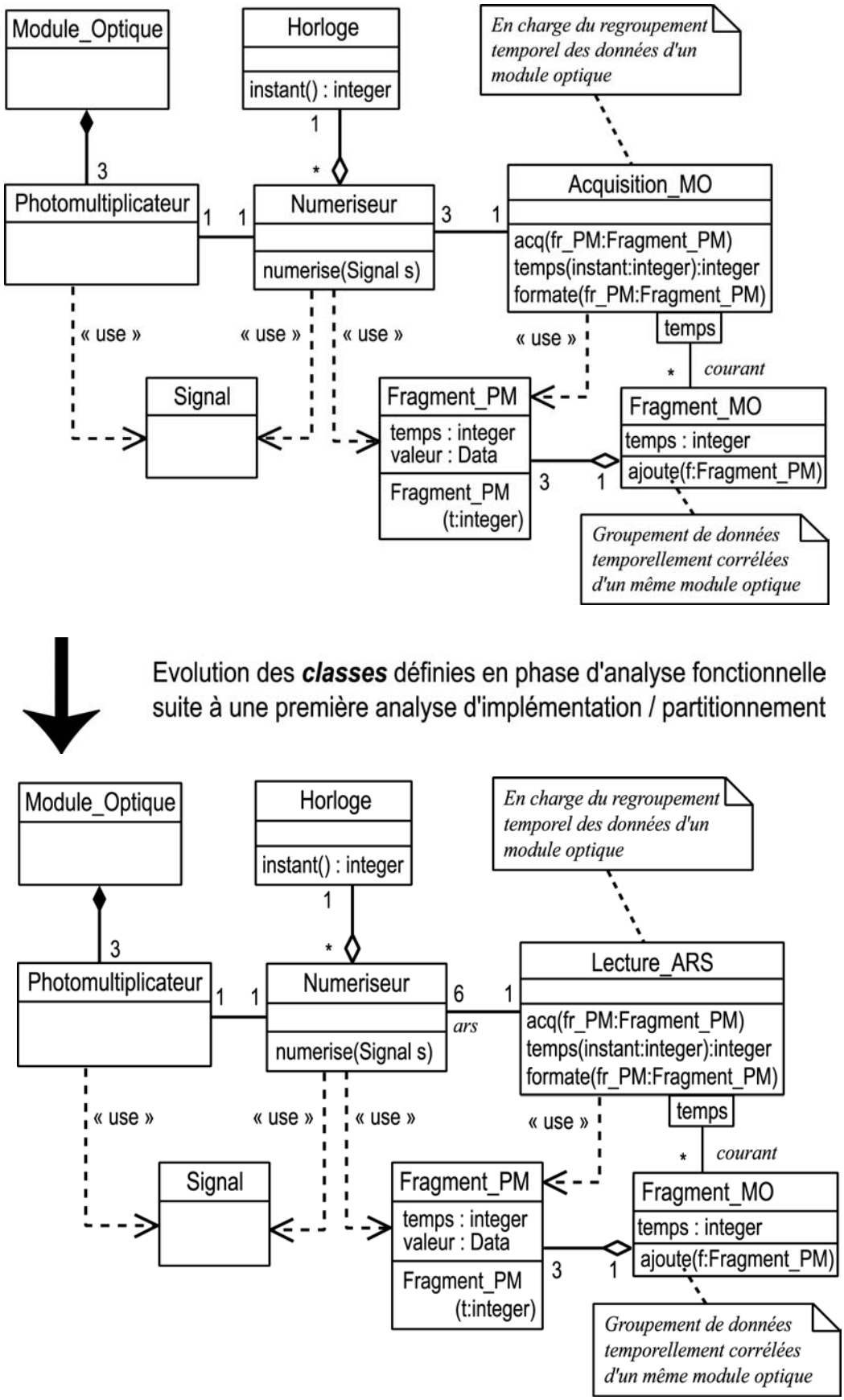


Fig. IV-13 : Evolution du diagramme de classe des objets intervenant dans la numérisation

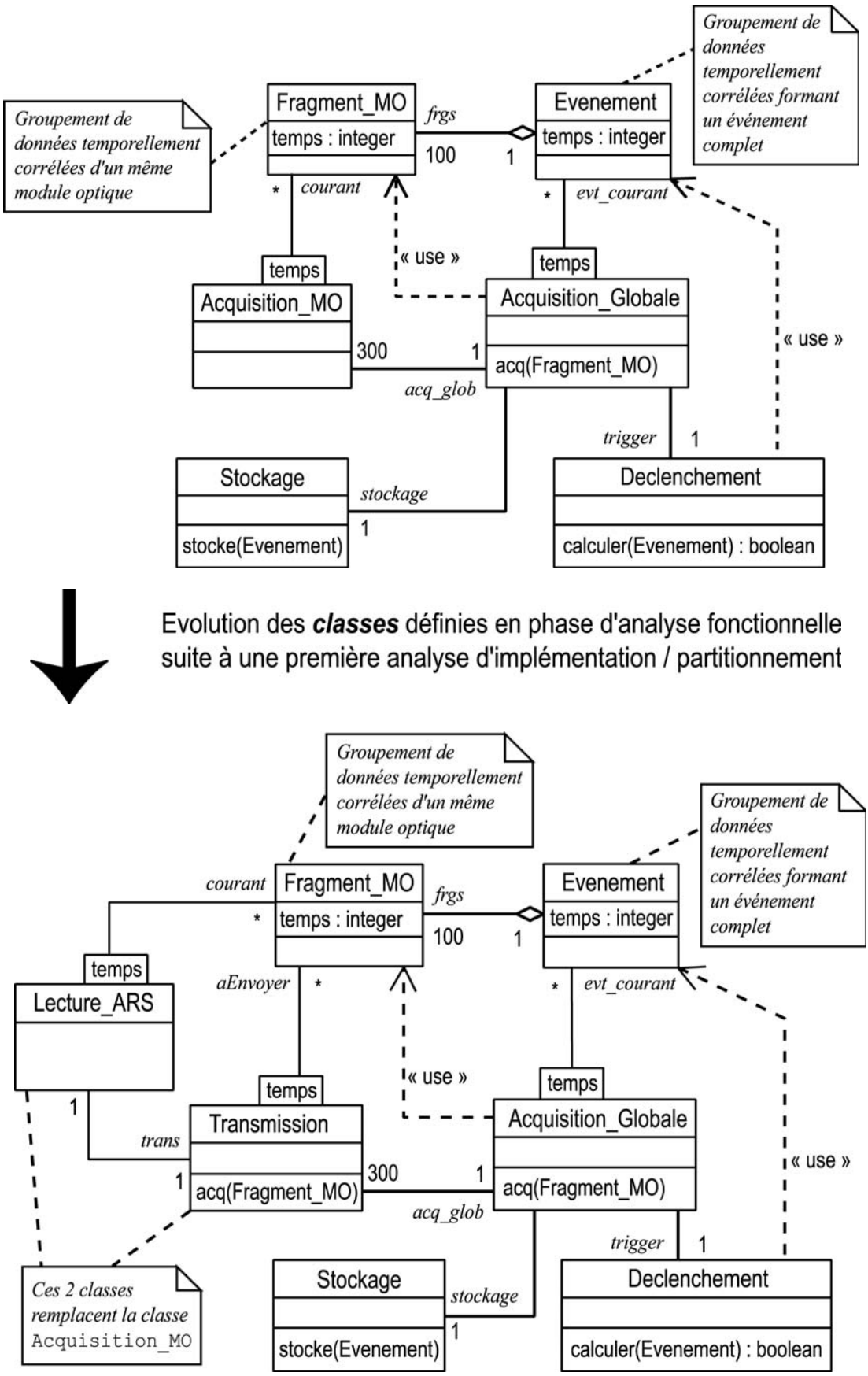


Fig. IV-14 : Evolution du diagramme de classe des objets intervenants dans l'acquisition globale

Des évolutions analogues doivent également s'appliquer aux diagrammes de classes des figures IV-8 et IV-9. Les figures IV-13 et IV-14 montrent respectivement ces évolutions. Nous voyons ainsi en figure IV-13 que l'on a, d'une part, remplacé la classe `Acquisition_MO` par `Lecture_ARS` et d'autre part multiplié par deux le nombre d'objets `Numeriseur` associés à `Lecture_ARS` (puisque chaque PM doit être numérisé par une paire d'ARS, cf. p. IV-20). Sur la figure IV-14, nous voyons que, conformément à notre analyse, la classe `Acquisition_MO` est remplacée par le couple `Lecture_ARS` et `Transmission`.

Notre analyse d'implémentation permet également de disposer d'un premier modèle d'implémentation global des objets du système de déclenchement / acquisition de notre exemple. Le diagramme de déploiement de la figure IV-15 représente ce modèle. Sur ce diagramme, les nœuds représentent les plateformes de mise en œuvre des différents objets spécifiés par la description fonctionnelle du système. Dans chaque nœud, différentes classes de l'application sont implémentées par l'intermédiaire d'un ou plusieurs composants. Les nœuds sont reliés entre eux par des liens de communication éventuellement monodirectionnels (comme entre les nœuds `FPGA` et `Processeur`). Le nom du lien de communication est une première indication sur sa nature. Il sera amené à être précisé au cours du processus de développement. Par exemple, le lien entre les nœuds `Processeur` et le nœud `Station` s'appelle ici `Reseau` car nous ne préjugeons pas encore du mode de communication précis entre ces nœuds : nous savons simplement qu'une forme de réseau les reliera nécessairement. Si nous choisissons plus tard un réseau Ethernet ainsi que, par exemple, l'usage du protocole UDP, nous pourrons renommer ce lien « UDP » ou « UDP/IP/Ethernet ». Nous verrons au IV.C comment utiliser ce nom pour désigner un modèle de communication formellement défini. Notons enfin que nous envisageons l'usage, au niveau des diagrammes de déploiement, de contraintes non formelles sur l'environnement matériel dont l'expression peut néanmoins orienter de manière déterminante le processus de développement. Les contraintes `{basse consommation}`, `{mono-composant}` et `{min: 21 Mb/s}`, respectivement sur les nœuds `Processeur`, `FPGA` et le rôle `etage` en sont des exemples. Les deux premières résultent des contraintes embarquées fortes liées à la nature et à l'environnement physique de l'expérience ANTARES (Cf. spécifications p. IV-13 ainsi que II.C.2, p. II-19) qui exigent une optimisation poussée de la fiabilité et de la dissipation

thermique des modules électroniques du système ; la troisième résulte directement de l'analyse des spécifications fonctionnelles lors du partitionnement matériel / logiciel (p. IV-21).

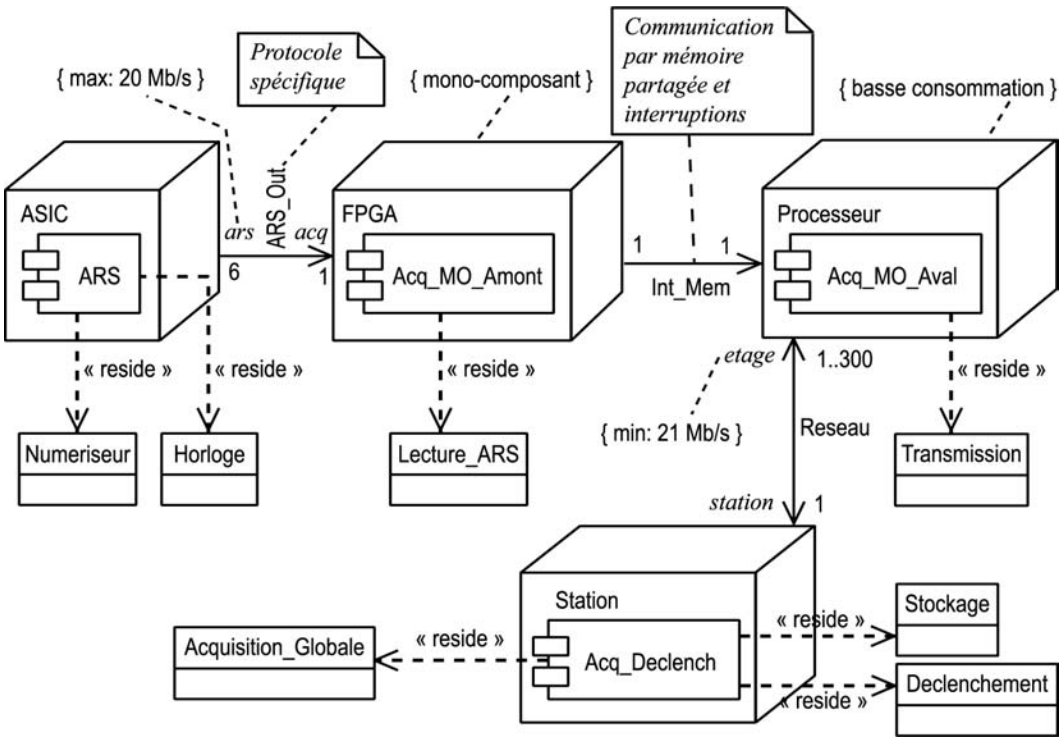


Fig. IV-15 : Premier modèle d'implémentation du système TDAQ d'ANTARES

IV.B.4.2 Compléter l'architecture du substrat matériel

Nous avons déjà évoqué p. IV-18 la notion de « substrat matériel » du système TDAQ. Il s'agit du squelette matériel du système sur lequel les composants matériels et logiciels de plus haut niveau seront mis en œuvre. L'esquisse de ce substrat est imposée par les caractéristiques intrinsèques du système, liées aux principes de détection envisagés. Nous avons vu par exemple au IV.A.1.1, p. IV-4 que la répartition géographique des modules de détection de l'expérience ANTARES était imposée par la nécessité scientifique. Nous savons donc que nous devrons acheminer les données numérisées dans chaque étage à travers un réseau dont la répartition géographique des nœuds (les étages de détection) et leur multiplicité est imposée. Cette donnée, associée aux contraintes naturelles liées à l'expérience vont nous permettre de compléter l'architecture du squelette matériel du système.

Nous pouvons par exemple exclure d'office toutes les technologies de réseau hertziens puisque les ondes hertziennes ne peuvent se propager dans l'eau. Par

ailleurs, nous savons que l'ensemble du câblage d'une ligne de détection sera contenu dans le câble électromécanique de cette ligne qui est lui-même limité en diamètre (Cf. contraintes p. IV-12). Sachant que les liaisons optiques occupent bien moins de place que les liaisons cuivre, il devient clair que le réseau devra être entièrement basé sur fibre.

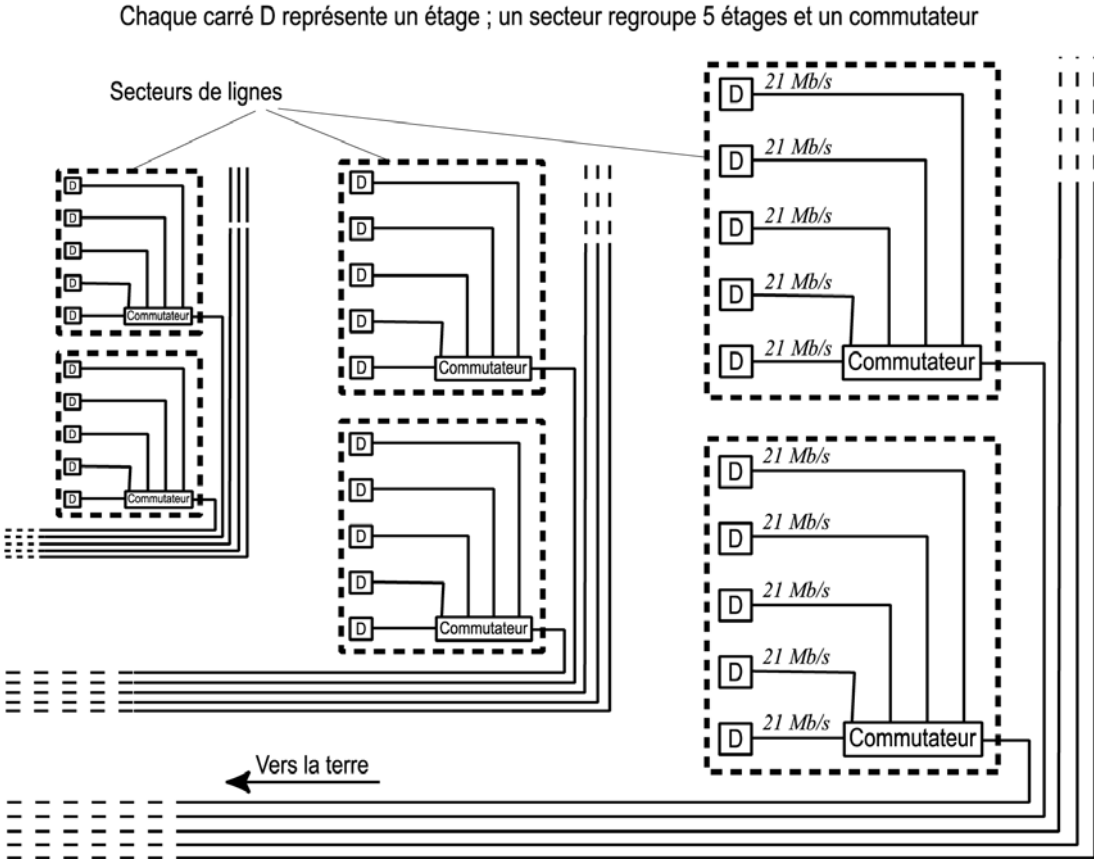


Fig. IV-16 : Substrat matériel du système d'acquisition offshore d'ANTARES (décomposition en secteurs des lignes de détection)

Comment établir la topologie du réseau ? Là encore, l'analyse de la répartition intrinsèque des étages de détection et des contraintes de l'application nous permet de conclure. En effet, l'impératif de fiabilité exige que le système soit le plus tolérant aux pannes possible. Il est donc en particulier souhaitable qu'une panne sur la liaison réseau d'un étage ne se propage pas aux autres étages. Par conséquent, l'idéal serait que chaque étage dispose d'une liaison propre jusqu'aux commutateurs situés à terre. Cependant, un tel dispositif se heurte encore à la limitation en diamètre du câble électromécanique ainsi qu'à la limitation du câble sous-marin reliant la terre aux lignes de détection. Le compromis qui fiabilise au maximum le réseau tout en respectant ces limitations correspond à une topologie « en arbre » : chaque ligne de 30 étages est décomposée en 6 secteurs de 5 étages

chacun. Chaque étage disposera de sa propre liaison optique jusqu'à la terre. L'ensemble des données émises par les 5 étages d'un même secteur devront donc être regroupée à travers un commutateur à 6 ports (5 entrées, et une sortie pour l'essentiel du flot de données) afin d'être acheminées jusqu'à la terre par une unique liaison optique.

L'analyse des contraintes naturelles à l'expérience nous conduit ainsi à établir la forme et les caractéristiques du substrat matériel de notre système TDAQ : la figure IV-16 résume graphiquement nos conclusions.

IV.B.4.3 Choisir les technologies matérielles de la partie matérielle

Comme annoncé p. IV-20, le partitionnement matériel/logiciel doit s'accompagner d'un certain nombre de choix en matière de technologies matérielles, aussi bien pour la mise en œuvre de la partie matérielle que de la partie logicielle.

En ce qui concerne les objets de la partie matérielle, les FPGA* représentent la première solution technologique envisagée. En effet, les FPGA permettent d'implémenter des circuits électroniques de hautes performances tout en offrant par leur caractère reprogrammable une très grande souplesse. L'usage généralisé des FPGA et du langage VHDL* dans la conception des cartes électroniques pour les systèmes TDAQ a grandement facilité de déverminage et la simulation des circuits en permettant de mener à bien l'essentiel de ces activités à l'aide de logiciels spécialisés. Il a notamment permis d'abaisser grandement les coûts des développements électroniques en rendant possible une conception haut niveau des circuits et en réduisant fortement les temps de développement ainsi que le nombre de prototypes avant le lancement de la fabrication en série. L'industrie produit à un rythme accéléré des composants FPGA intégrant de plus en plus de cellules élémentaires ce qui autorise l'implémentation de circuits de plus en plus complexes sur un seul composant. Pour les modules qui nécessitent des performances extrêmes ou des très grandes séries, on peut envisager de les implémenter sur des ASIC*, ce qui implique un cycle de développement beaucoup plus lourd et des coûts de conception et de production de prototypes beaucoup plus élevés. Il est à noter que le développement d'un ASIC comprend souvent une phase de développement d'un prototype sur FPGA.

Le choix de la technologie matérielle pour les objets de la partie matérielle au

cours de l'étude d'implémentation de système TDAQ consistera donc essentiellement à déterminer pour chaque module s'il sera mis en œuvre sur des FPGA ou des ASIC, les critères principaux étant essentiellement (sinon exclusivement) le coût global et les performances. Il s'agira ensuite d'affiner ce choix en étudiant l'offre des industriels en matière de composants. Par exemple, les grands fabricants de FPGA tels Altera [A-21] ou Xilinx [A-22] se distinguent assez fortement par la structuration interne de leurs composants, ce qui a le plus souvent une incidence non négligeable sur leur adéquation à l'application TDAQ en question. Ainsi, dans l'étude de l'acquisition offshore de l'expérience ANTARES, l'implémentation de l'objet en charge de la lecture des ARS (Cf. figures IV-12 et IV-13) a été envisagée sur des composants Altera et Xilinx ; une étude détaillée de l'architecture de l'objet a permis d'aboutir à une décision sans équivoque en faveur du second[7].

IV.B.4.4 Choisir les technologies matérielles de la partie logicielle

Le choix de la plateforme matérielle pour l'implémentation de la partie logicielle est plus complexe que celui concernant la partie matérielle car l'offre disponible est beaucoup plus pléthorique et les technologies beaucoup lus variées. Il s'agira donc pour les ingénieurs concernés de maintenir une connaissance vaste et approfondie du marché par une activité de veille technologique poussée. Les critères de choix incluent ceux qui président à la partie matérielle, à savoir les performances matérielles recherchées, la fiabilité, la consommation électrique, etc. S'ajoutent à ceux-ci des critères d'interconnexion avec la partie matérielle, comme par exemple la possibilité pour tel processeur de communiquer à travers tel medium matériel, ainsi que des critères spécifiquement logiciels comme la disponibilité et/ou la performance de tel système d'exploitation ou de tel *middleware** sur la plateforme matérielle envisagée. Par exemple, dans la première phase de l'expérience NA48, les contraintes de latence étaient telles qu'il était indispensable de recourir à une plateforme multiprocesseurs autorisant de nombreux liens de communication à latence faible aussi bien entre les nœuds de calcul qu'avec les modules électroniques en amont du flot de données ; le choix s'est finalement porté sur des cartes VME* à huit processeurs DSP* interconnectés par des liens point-à-point rapides selon une topologie assez dense [C-1] [B-8].

7. *Nous ne détaillerons pas plus le cheminement qui a mené à cette décision car cela nous contraindrait à discuter en détail des différences d'architecture interne entre FPGA, ce qui nous mènerait largement hors du cadre de cette thèse.*

Pour avoir un aperçu de l'analyse qui oriente le choix de la plateforme matérielle pour la partie logicielle, considérons à nouveau notre exemple inspiré d'ANTARES.

IV.B.4.4.a Choix matériel pour le sous-système offshore d'ANTARES

Nous avons vu (Cf. fig. IV-15) que le processeur de l'acquisition offshore (nœud `Processeur`) doit être un composant à consommation électrique réduite, que la communication entre les étages de détection et les stations à terre doit nécessairement s'appuyer sur une forme de réseau dont les performances doivent permettre un débit sortant d'au moins 21 Mb/s effectifs au niveau de chaque étage d'acquisition. Ajoutons également que les limitations mécaniques imposées au système ont conduit la collaboration ANTARES à décider que les processeurs de l'acquisition offshore assureront également le contrôle lent des étages de détection, c'est-à-dire que le logiciel de contrôle lent devra tourner sur le nœud `Processeur` et accéder aux autre modules électroniques de l'étage à travers des liens série de type « bus de terrain ». Par conséquent, le processeur en charge de l'acquisition offshore doit pouvoir accéder à des ressources réseau ainsi qu'à des liaisons série. Or, l'un des facteurs les plus importants dans la dégradation de la fiabilité d'une carte électronique est de nombre de composants (puces) qu'elle comporte. En d'autres termes, plus les fonctionnalités d'une même carte sont intégrées sur un nombre réduit de puces, meilleure est la fiabilité de cette carte[8].

L'étude de l'offre des industriels en matière de processeurs destinés au marché de l'embarqué permet de constater qu'il existe des processeurs intégrant l'ensemble de ces caractéristiques. La puce choisie pour l'expérience ANTARES est le MPC860P de la famille PowerPC fabriqué par la société Motorola [B-28]. Elle comprend, outre un cœur PowerPC pouvant fonctionner à 80 MHz, un contrôleur de SDRAM*, 4 ports série, un port Ethernet* à 100 Mb/s ou ATM* à 155 Mb/s. Sa consommation électrique nominale est de l'ordre de 800 mW. Cette intégration poussée et sa consommation très réduite la destinent à des applications basse consommation et haute fiabilité. De nombreux systèmes d'exploitation industriels temps réel embarqués ont été portés sur les plateformes à base de MPC860.

Notons que le port réseau de la puce peut être de type ATM ou Ethernet, ce qui

8. *Cela est principalement dû au fait que les interconnexions sur le substrat semiconducteur d'une même puce sont plus fiables de plusieurs ordres de grandeur que celles mises en œuvre sur le PCB d'une carte électronique.*

laisse le choix libre quant à la technologie du réseau offshore. Or, l'activité d'assemblage d'événement* (Cf. II.A.3) entraîne des flux de données instantanés de plusieurs sources vers une seule destination, ce qui est générateur de congestion réseau [C-4] [C-31]. ATM permettrait une gestion rigoureuse de la bande passante au niveau de la couche réseau ce qui faciliterait le contrôle de la congestion. Mais il faut garder à l'esprit que la totalité du flux des données d'acquisition doit être commutée à terre pour effectuer l'assemblage des événements*. Or, d'une part, les coûts des commutateurs ATM sont bien plus élevés que ceux de la norme Ethernet, et d'autre part, l'avenir industriel de la norme ATM demeure incertain. Aussi, malgré l'intérêt technique présenté par ATM, la collaboration ANTARES a opté pour la technologie Ethernet.

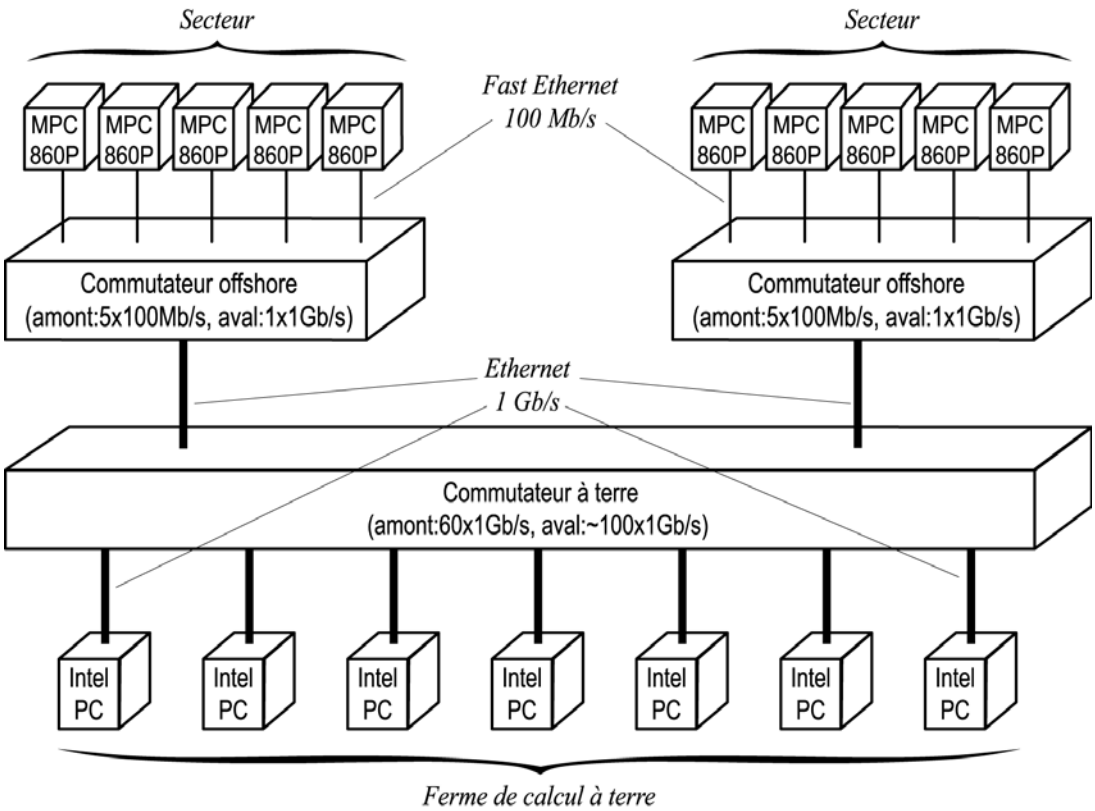


Fig. IV-17 : Architecture réseau et implémentation matérielle de la partie logicielle (exemple d'ANTARES)

IV.B.4.4.b Choix matériel pour le sous-système à terre d'ANTARES

En ce qui concerne le matériel qui supportera l'acquisition et le traitement des données à terre, le choix est relativement simple car les contraintes de l'embarqué ne s'y appliquent plus. Nous avons besoin de puissance de calcul et d'évolutivité tout en maintenant des coûts bas, ce qui nous conduit naturellement vers des stations de calcul, éventuellement multiprocesseurs, à base de processeurs Intel ou

AMD. Le flot de données total sera important puisqu'égal en moyenne à 21 Mb/s × 300 étages = 6.3 Gb/s. Il semble donc clair que nous devrons recourir à terre à du parallélisme de performance : l'assemblage d'événements et les calculs de déclenchement à terre devront donc s'effectuer sur une ferme de stations. Leur connexion au réseau devra très certainement être du type Gigabit Ethernet*.

IV.B.4.4.c L'architecture matérielle de la partie logicielle du système

Après cette analyse, nous pouvons voir que la partie logicielle de notre système TDAQ devra être mise en œuvre sur un réseau Ethernet commuté de plusieurs centaines de nœuds. les nœuds offshore seront implémentés sur des processeurs basse consommation à haut niveau d'intégration tels le MPC860P alors que les nœuds à terre constitueront une ferme de calcul basée sur des stations au rapport performances / prix élevé. L'architecture matérielle de la partie logicielle de notre système est résumée sur la figure IV-17.

IV.B.4.5 Choisir les technologies logicielles de la partie logicielle

Le choix des technologies logicielles à employer pour le développement de l'application TDAQ est, bien entendu, plus complexe que les choix concernant les supports matériels, car non seulement l'offre est pléthorique mais les critères à prendre en compte sont nombreux et souvent difficiles à évaluer. Parmi ces critères, l'adéquation aux choix matériels est déterminante[9]. Par exemple, l'usage d'un processeur basse consommation implique des limites fortes sur la puissance de traitement et la bande passante de la mémoire, ce qui exclut les technologies logicielles trop consommatrices des ressources correspondantes. Une autre source de difficultés est le caractère hétérogène du système TDAQ. Ainsi que nous l'avons vu en IV.B.4.4, les différents sous-systèmes du détecteur présentent des contraintes et des exigences le plus souvent différentes, ce qui nous mène immanquablement vers une mise en œuvre matérielle hétérogène. Or, au moins au niveau de l'application, ces hétérogénéités devront être bien entendu gommées.

En résumé : étant donné une plateforme matérielle hétérogène et fortement distribuée, il s'agit de choisir les technologies et les briques logicielles qui nous

9. En fait, comme évoqué p. IV-30, le choix de la plateforme matérielle est déjà lui-même lié aux possibilités qu'elle offre en termes de technologies logicielles. Aussi faut-il toujours garder à l'esprit que notre cycle de développement TDAQ est de nature incrémentale, les choix à chaque étape ayant a priori *une incidence sur toutes les autres étapes du cycle sous la forme d'une boucle de rétroaction (cycle en spirale).*

permettront de développer une application TDAQ temps réel robuste, performante et maintenable. Les choix récurrents concernent trois volets principaux :

- Les langages de programmation
- Les systèmes d'exploitation
- Les *middlewares*[*]

IV.B.4.5.a Choix des langages de programmation

La problématique du choix d'un (ou plusieurs) langage(s) de programmation pour le développement d'un système TDAQ peut paraître artificielle ou désuète. En réalité, l'acquis historique en physique nucléaire et en physique des particules aboutit aujourd'hui à une situation très particulière dont il faut tenir compte dans tout processus de développement des futurs systèmes TDAQ. Nous avons notamment vu au II.C.3 que pour diverses raisons, le modèle objet rencontre des réticences de la part d'une frange des physiciens HEP[*], principalement à cause d'une perception négative du langage C++. De plus le portage des outils d'analyse et de simulation scientifiques sur des plateformes objet (essentiellement en C++) n'est, à ce jour, pas encore finalisé. Bien entendu, le choix d'un langage de programmation ne doit se faire que sur des critères purement techniques fondés sur la disponibilité et la qualité des compilateurs, les performances attendues ainsi que la maturité industrielle du langage et des outils associés. Ces critères excluent clairement le FORTRAN dès que nous nous plaçons dans le domaine de l'embarqué et du temps réel, les seuls candidats plausibles étant aujourd'hui que le C, le C++ et, dans une certaine mesure, Java. Par conséquent, si le système développé entretient une interaction forte avec les codes d'analyse scientifique[10] et que ceux-ci reposent encore sur le FORTRAN, il s'agira de définir des interfaces simples et claires entre le code TDAQ et le code d'analyse, incluant éventuellement des mécanismes de liaison avec des programmes en FORTRAN. Un contre-exemple est le cas de la simulation du sous-système de déclenchement des chambres à dérive de l'expérience NA48 : le code de déclenchement temps réel (sur DSP, puis sur processeurs RISC) étant écrit en C, la simulation correspondante dans les codes d'analyse était une réécriture en FORTRAN de l'algorithme, ce qui a nui (du moins au départ) à la garantie d'identité statistique entre les deux codes (il n'était pas évident que les biais statistiques introduits par les deux codes fussent identiques).

10. Ce qui est le cas lorsqu'il s'agit d'un système de déclenchement, cf. II.A.3

Cependant, l'exemple d'ANTARES montre que ces difficultés spécifiques à la physique des hautes énergies tendent à disparaître grâce à l'adaptation des techniques de travail et l'évolution des mentalités. D'une part, aussi bien le code de déclenchement temps réel sur les stations à terre que celui du système d'acquisition offshore (fig. IV-17) sont développés en C++. L'alternative Java, si elle demeure envisageable pour le code de déclenchement, est pour l'instant exclue en ce qui concerne la partie temps réel embarquée en raison d'un manque de maturité industrielle dans ce domaine. Les codes de simulation et d'analyse sont encore pour l'essentiel en FORTRAN car ils utilisent les canevas éprouvés du domaine HEP*, mais ils sont développés dans un cadre où les physiciens connaissent les différents langages et s'y adaptent en fonction des nécessités attachées à chaque module. Dans cette perspective, il est clair que, par exemple, le code de déclenchement temps réel sera utilisé tel quel au sein du canevas d'analyse global, ce qui garantira l'identité de comportement statistique entre les traitements simulés et les traitements réels.

IV.B.4.5.b Choix des systèmes d'exploitation

La problématique liée au choix du système d'exploitation pour une plateforme matérielle donnée s'articule autour de trois points principaux :

- Les facilités de développement, d'intégration et d'évolution
- Les performances (CPU, réseau, temps réel)
- Le coût (licences de développement et exécutifs)

Le but de notre méthodologie étant de maximiser la qualité des systèmes réalisés par une rationalisation du processus de développement, le premier critère sur les facilités de développement est celui que les développeurs doivent garder avant tout à l'esprit. Sacrifier à ce critère afin d'obtenir de meilleures performances ou des coûts inférieurs ne devrait se faire qu'en cas de réelle nécessité. A ce titre, les systèmes d'exploitation qui présentent une interface système « standard » (par exemple POSIX) doivent être favorisés dans la mesure du possible afin de faciliter le portage multi-plateformes des applications.

L'expérience prouve que les performances liées à un système d'exploitation, et notamment aux pilotes logiciels de la plateforme considérée, ne peuvent réellement être évaluées que par des tests réels. Ils n'entrent donc en ligne de compte

dans le choix du système d'exploitation qu'à partir de la seconde itération dans le cycle de développement TDAQ, c'est-à-dire après une première série de tests sur des dispositifs d'évaluation (fig. IV-18). Avant cela, les seuls critères de performances que l'on puisse considérer sont le caractère réellement temps réel (à latence maîtrisée) ainsi que l'empreinte mémoire, critères pour lesquels on peut obtenir une réponse fiable sur la base des données des vendeurs.

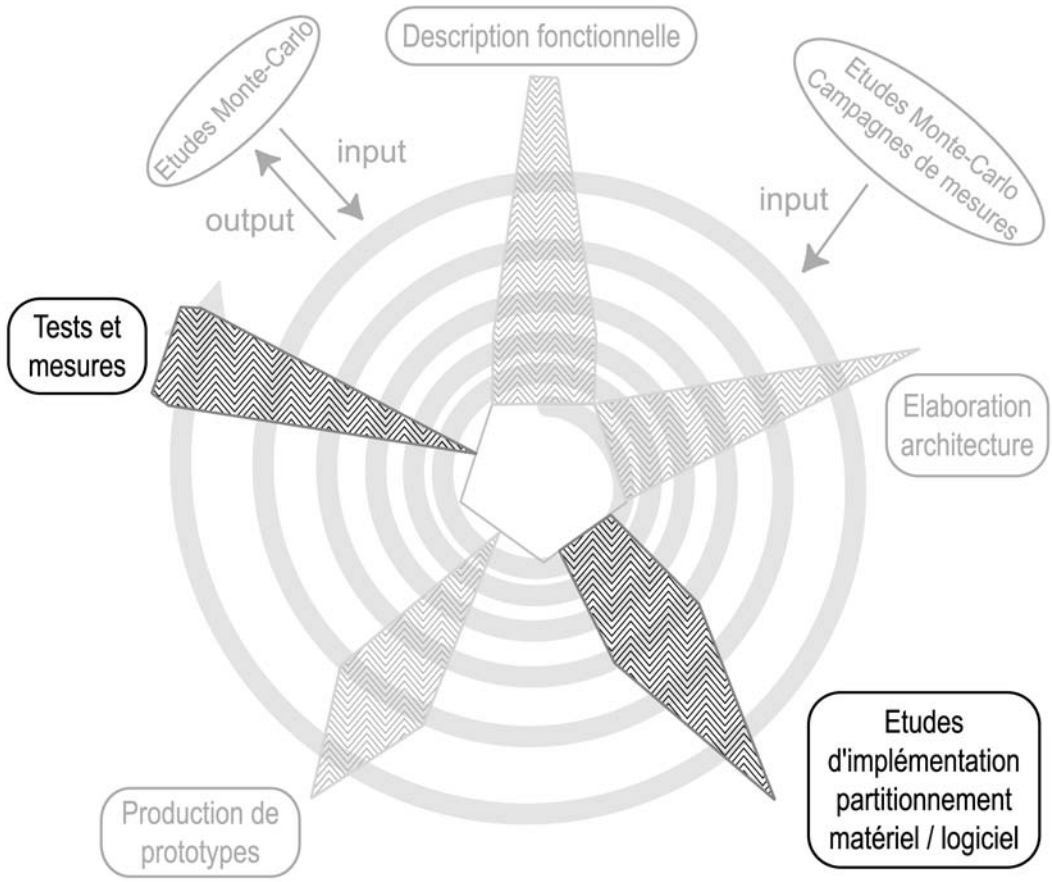


Fig. IV-18 : Phase d'implémentation/partitionnement du cycle de développement TDAQ

La question des coûts associés au système d'exploitation est fortement dépendante de l'application TDAQ considérée. Dans le sous-système de déclenchement de l'expérience NA48, par exemple, le nombre total de processeurs concernés est de l'ordre de la douzaine ce qui implique que l'essentiel du coût provient des licences de développement. En revanche, le projet « 0.1 km^2 » d'ANTARES comporte quelque 300 processeurs uniquement dans la partie offshore, ce qui multiplie d'autant le coût de l'exécutif et rend, par exemple, les solutions « sans *runtime* » attractifs.

La question des coûts liés au système d'exploitation a pris un nouveau visage depuis le développement de Linux, d'autant plus que le succès du système

d'exploitation libre dans l'enseignement et la recherche a été particulièrement fulgurant puisque les stations de travail et les grappes de calcul passent sous Linux à un rythme accéléré. Linux étant également porté (dans des versions plus ou moins modifiées) sur une multitude de plateformes, dont certaines temps réel ou embarquées, le choix quasi-systématique de ce système pour les applications TDAQ devient une option envisageable. En effet, l'adoption du « tout Linux » dans les expériences de physique aurait l'immense avantage d'homogénéiser l'environnement de développement de toutes applications depuis l'analyse hors ligne jusqu'aux codes d'acquisition dans les processeurs enfouis. Une telle décision va complètement dans le sens de notre premier critère de choix de système d'exploitation (facilité de développement, d'intégration et d'évolution) tout en satisfaisant parfaitement le critère de coût puisqu'alors, les seuls coûts non négligeables correspondraient à un éventuel support extérieur, ce qui est de toutes façons également payé pour les systèmes d'exploitation propriétaires.

Aujourd'hui, l'adoption de Linux sur plateforme compatible Intel pour tout ce qui concerne les fermes de calcul dans les systèmes TDAQ est une évidence en termes de performances, de coût, de disponibilité d'outils de développement et d'évolutivité matérielle et logicielle. Pratiquement toutes les expériences de physique des hautes énergies s'orientent vers cette solution pour leurs fermes de calcul (même s'il subsiste encore des développements sous Windows NT). A ce titre, l'exemple d'ANTARES n'est pas une exception et le choix de cette solution pour la mise en œuvre des objets :`Stockage`, `:Acquisition_Globale` et `:Declenchement` (fig. IV-15) sur le(s) nœud(s) `Station` est limpide. La possibilité d'adopter également Linux pour les processeurs offshore MPC860P par souci d'uniformisation a donc été sérieusement envisagée. Cette possibilité a cependant été –pour l'instant– rejetée en faveur du système d'exploitation temps réel VxWorks pour des questions de performances (Cf. IV.B.6) et ce, malgré un coût supérieur. De plus, le processeur offshore devant accueillir le processus de contrôle lent en plus de l'acquisition (Cf. IV.B.4.4.a), il est préférable que le système d'exploitation respecte strictement les priorités afin que le partage des ressources CPU entre les deux processus ainsi que la gigue sur la latence des commandes du contrôle lent soient correctement contrôlés.

IV.B.4.5.c Choix des middlewares*

Depuis le début des années 1990, l'utilisation de *middlewares* pour le développement d'applications TDAQ devient une pratique de plus en plus répandue. Un domaine où l'usage d'un *middleware* s'impose clairement est celui des communications réseau. Il semble entendu qu'il soit plus économique en temps et en efforts de baser les communications de l'application TDAQ sur une couche de communication mettant directement en œuvre des concepts de haut niveau que de réinventer (et redévelopper) cette couche à chaque fois. Il n'en demeure pas moins, comme nous le verrons sur nos exemple, que le choix du *middleware* doit s'appuyer sur des critères suffisamment stricts pour que ce qui est censé épargner aux développeurs du travail inutile n'aboutisse pas *in fine* à les enfermer dans un carcan. Nous distinguons 4 questionnements autour desquels l'adoption d'un *middleware* pour une application TDAQ doit s'articuler :

- Quels sont les bénéfices attendus de l'emploi du *middleware* ?
- Le *middleware* correspond-il à un standard ou une norme ?
- Quels sont les modalités de distribution du *middleware* (sources libres, …) ?
- Quels en sont les coûts en performances (CPU, mémoire, réseau) ?

Ces cinq questionnement sont loin d'être indépendants les uns des autres et nous allons voir qu'elles s'influencent en fait assez fortement.

Le première question est celle à laquelle il faut répondre en premier afin de déterminer l'utilité réelle à employer le *middleware* envisagé. En effet, l'usage d'un *middleware* implique, outre un éventuel investissement financier, un investissement intellectuel correspondant au temps et aux efforts nécessaires à un apprentissage efficace du produit. Il faut donc pouvoir raisonnablement espérer que cet effort soit compensé par un gain de temps et d'efforts supérieur, sinon sur le développement de l'application, du moins sur le développement d'applications ultérieures.

La réponse à cette dernière possibilité (gain sur les applications ultérieures) nous amène à la deuxième question car un tel gain à n'est réellement envisageable que si le produit correspond à la mise en œuvre d'un standard ou d'une norme. Par exemple, si nous décidons d'employer un *middleware* CORBA* pour assurer la communication inter-objets dans notre application, nous savons que l'investisse-

ment intellectuel aura de grandes chance d'être payant car la norme CORBA est un standard (Cf. [B-13]) ; nous pourrons donc continuer à nous appuyer dessus au-delà de l'application présente tout en demeurant relativement indépendants du *middleware* choisi car la norme définit des API* ainsi que des formats d'interopérabilité entre ORB*. Le cas inverse est celui où le *middleware* met en œuvre une architecture et des interfaces propres, ne correspondant ni à une norme, ni à un standard, auquel cas, il serait plus prudent de viser des gains pour l'application en cours. Les *middlewares* de communication par messages tels ControlHost [B-30], par exemple, impliquent un investissement intellectuel suffisamment léger pour en envisager l'usage dans le cadre d'une seule application.

Cependant, même dans ce dernier cas (investissement intellectuel faible), la troisième question des modalités de distribution du *middleware* ne doit pas être négligée, essentiellement en raison de du temps de vie long des expériences de physique. En effet, une distribution purement binaire d'un composant logiciel propriétaire entraîne une dépendance envers un vendeur particulier, ce qui peut être préjudiciable si le vendeur disparaît ou s'il ne supporte plus son produit. A ce titre, l'exemple de l'expérience NA48 est éloquent. En effet, la collaboration NA48 avait décidé d'utiliser un *middleware* de communication commercial appelé Isis[11]. Les développements les plus en avance tels celui du sous-système de déclenchement des chambres à dérive, se sont donc poursuivis en utilisant massivement les librairies Isis. Plus tard, la société éditrice de Isis a fait faillite alors que les développements du sous-système étaient presque terminés. En l'absence de ressources humaines suffisantes pour porter le code sur un autre *middleware*, l'application est restée dépendante du code binaire propriétaire de Isis, ce qui la lie également à un système d'exploitation et une plateforme matérielle particuliers (SunOs 4 sur Sparc). Depuis, toutes les tentatives d'amélioration matérielle ou logicielle du système se heurtent à cette barrière. Un cas assez parlant est celui du passage à l'an 2000 : SunOS 4 « ne passe pas l'an 2000 » au sens où au 1er janvier 2000, la date du système revient au 1er janvier 1970, ce qui entraîne un marquage temporel des fichiers produits erroné. La solution serait de passer à Solaris, mais Isis ne fonctionne pas sur Solaris car celui-ci est trop en rupture avec SunOS. La « solution » adoptée a donc été de fausser la date système en la reculant de quelques années et en en tenant compte pour les nouveaux fichiers produits[12].

11. Aucune référence n'est disponible en raison de la disparition du middleware.

Nous voyons donc que si l'on opte pour un *middleware* qui ne correspond pas à un standard ou à une norme, le risque de se heurter à des problèmes d'évolutivité est grand, sauf si l'on a la maîtrise de son code source. En effet, les *middlewares* distribués en « *open source* » présentent moins de risques car leur recompilation sous de nouvelles plateformes est possible. ControlHost [B-30], par exemple, est également un *middleware* de communication par messages développé en *open source* par des ingénieurs du domaine HEP. Bien qu'il soit doté de fonctionnalités moins puissantes par rapport à Isis, sa simplicité et la disponibilité de ses sources en ont fait un *middleware* utile et évolutif pour les systèmes TDAQ, ainsi que le prouve son usage dans l'expérience CHORUS [A-18] ou dans le sous-système de contrôle global de l'expérience NA48. Ces qualités ont été également confimées lorsque la collaboration ANTARES a décidé de l'utiliser dans un premier temps pour la transmission des données et des messages. En effet, le code correspondant au client ControlHost a été facilement porté sur la plateforme offshore constituée par le processur MPC860P et le système d'exploitation temps réel VxWorks (Cf. IV.B.4.5.b et fig. IV-17).

Un autre exemple de *middleware* envisagé pour le système d'acquisition d'ANTARES est celui de ACE [A-20] [B-22]. Cet exemple est intéressant car il illustre bien les problématiques soulevées par la mise en œuvre et l'implémentation des systèmes TDAQ. ACE (*Adaptative Communication Environment*) est un *middleware open source* objet écrit en C++ destiné à unifier l'ensemble des concepts et motifs récurrents liés aux systèmes distribués multitâches sous un unique canevas multi-plateformes orienté objet. L'usage de ACE permet donc d'écrire des applications distribuées et multitâches très facilement portables sur de multiples plateformes[13] tout en bénéficiant des avantages du paradigme objet. ACE implémente également quelques motifs récurrents orientés objet relatifs à la distribution et la concurrence tels l'« objet actif » [C-11], ou le mécanisme « Accepteur-Connecteur » [C-12]. Ce *middleware* s'accompagne aussi de TAO (*The ACE ORB*) un ORB* CORBA* doté de spécificités adaptées pour le temps réel (tels des mécanismes d'extension de priorité sur le réseau). Ces différentes qualités font de ACE

12. Il est possible, si les budgets alloués le permettent, que le dispositif de l'expérience NA48 soit mis à jour et adapté pour une nouvelle expérience (KABES), auquel cas, des ressources suffisantes seraient dégagées pour procéder à une évolution majeure du code. Une première étude sur cette question va être effectuée à l'automne 2001.

13. De Linux à VxWorks, en passant par Solaris, QNX, etc.

un *middleware* très intéressant pour les applications TDAQ et donc, pour ANTARES en particulier. En revanche, comme souvent en matière de canevas orienté objet, l'investissement intellectuel qu'il nécessite est relativement important si l'on souhaite en maîtriser l'architecture interne ; il n'est donc réellement intéressant que s'il offre aux ingénieurs du système TDAQ d'ANTARES l'espoir d'être réutilisé pour d'autres expériences.

Enfin, une dernière question que l'on doit se poser autour de l'adoption d'un *middleware* est bien entendu celle des performances, en particulier si celui-ci doit être mis en œuvre au cœur d'un code d'acquisition ou de déclenchement. Il s'agit en effet de s'assurer que l'introduction du *middleware* n'entraînera pas la mise en place d'une architecture ou de couches logicielles qui seront préjudiciables aux performances du sous-système. L'usage de ControlHost pour la transmission des données dans ANTARES ne dégradera-t-il pas trop la bande passante utile ? Les librairies de ACE ne consommeront-elles pas trop de mémoire ?

Même si l'analyse *a priori* de l'architecture et des principes de conception du *middleware* est utile dans le traitement d'une telle question, il est clair que la réponse finale ne viendra que de tests réels effectués lors de la phase « Tests et mesures » du cycle de développement (Cf. fig. IV-18). Ce n'est donc qu'après une première itération du cycle que l'on pourra commencer à répondre à la question des performances du *middleware*.

IV.B.5 Production de prototypes

L'une des idées maîtresses du cycle de développement que nous proposons est le développement de systèmes prototypes très tôt dans le cycle, et ce afin de rendre possibles des tests de faisabilité et de performances qui seuls peuvent apporter des réponses fiables aux alternatives techniques qui se présentent aux développeurs. Nous avons ainsi vu au IV.B.4 que beaucoup de questions sur l'implémentation matérielle et logicielle du système TDAQ ne peuvent être résolues sans la contribution de tests sur prototypes.

Par « protoype », il faut donc entendre tout dispositif matériel et logiciel permettant de faire fonctionner, même partiellement, des processus liés à notre modèle d'implémentation du système. Rappelons également que notre modèle de développement des systèmes TDAQ préconise une croissance incrémentale du système

depuis les prototypes jusqu'au système réel par complexification progressive.

La figure IV-19 montre que l'activité de production de prototype est peu importante (voire nulle) en début de cycle de développement et croît progressivement à mesure que l'on avance dans les itérations du cycle, pour ensuite s'amenuiser jusqu'à annulation vers la fin du développement. La description fonctionnelle, l'élaboration d'architecture et l'étude d'implémentation nous fournissent après les premières itérations une vision suffisamment précise des fonctionnalités et des architectures matérielle et logicielle de principe pour que l'on puisse commencer à produire des dispositifs simples destinés à choisir, tester ou valider les solutions de mise en œuvre. Au cours du développement, ces dispositifs sont soit remplacés par d'autres soit modifiés et complétés progressivement jusqu'à ce que l'on parvienne au système final, point où l'activité de production de prototypes s'arrête. En d'autres termes, les système TDAQ final est le « prototype » le plus complet produit au cours du développement.

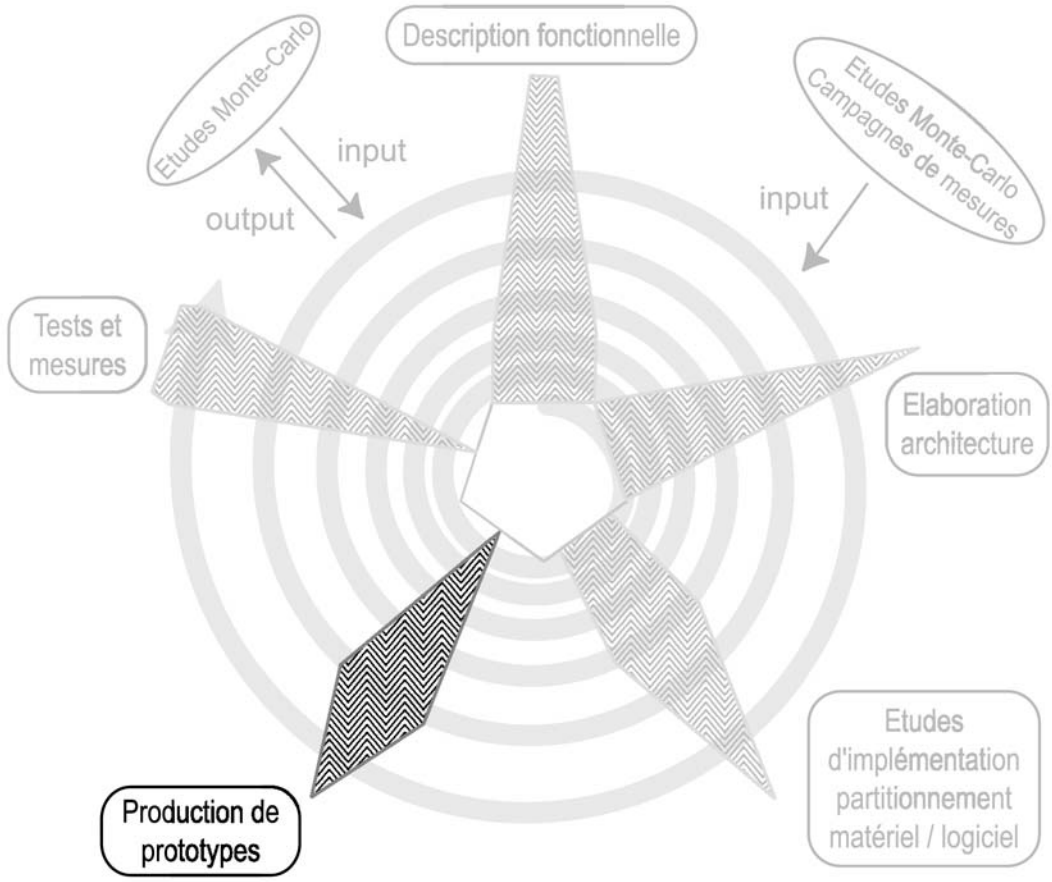


Fig. IV-19 : Phase de production de prototypes du cycle de développement TDAQ

Nous distinguons 3 objectifs principaux que la production de prototypes doit

permettre d'atteindre :

- Choisir entre plusieurs solutions d'implémentation
- Valider des architectures et dimensionner le système
- Tester des modules logiciels ou matériels développés

IV.B.5.1 Choisir entre plusieurs implémentations

Le but du prototype est dans ce cas de pouvoir trancher entre plusieurs solutions ou technologies envisagée au cours des études d'implémentation.

Pour notre travail sur ANTARES, par exemple, nous avons vu (IV.B.4.5.b, p. IV-36) que le choix de Linux comme système d'exploitation offshore présentait beaucoup d'intérêt à condition que les performances soient à la hauteur de l'application et que son caractère non temps réel (du moins pour le Linux standard) ne présente pas d'inconvénient majeur. L'implémentation envisagée sur la figure IV-15 nous indique que le composant `Acq_MO_Amont` implémenté dans le FPGA devra pouvoir absorber des pointes de 120 Mb/s depuis l'ARS et stocker les données dans une mémoire partagée avec le `Processeur`. Le composant `Acq_MO_Aval` devra évacuer ces données depuis cette mémoire jusqu'à terre à travers un port Ethernet 100 Mb/s au rythme moyen de 21 Mb/s. Ce dernier chiffre est une grandeur moyenne qui provient d'une estimation fondée sur notre connaissance des photomultiplicateurs et du niveau de bruit dans l'environnement du détecteur. Il nous faut donc faire la conception en visant un facteur de sécurité au moins égal à 2, afin de tenir compte d'éventuelles erreurs et de possibles fluctuations dues au comportement global du réseau. Nous devons donc viser flux moyen en sortie du processeur d'environ 40 Mb/s. Le travail du composant `Acq_MO_Aval` étant essentiellement basé sur ce chiffre, nous somme ainsi conduits à mettre en place un dispositif permettant de tester cette capacité sur différents systèmes d'exploitation.

Ce dispositif doit comprendre une station émettrice basée sur le processeur MPC860P et au moins une station réceptrice sous Linux ; afin de s'assurer que le flux de données ne soit limité que par le MPC860P, il serait préférable d'avoir deux stations réceptrices et un commutateur entre eux et la station émettrice, comme indiqué en figure IV-20. De cette manière, le débit maximum mesuré correspondra aux capacités maximales du module offshore.

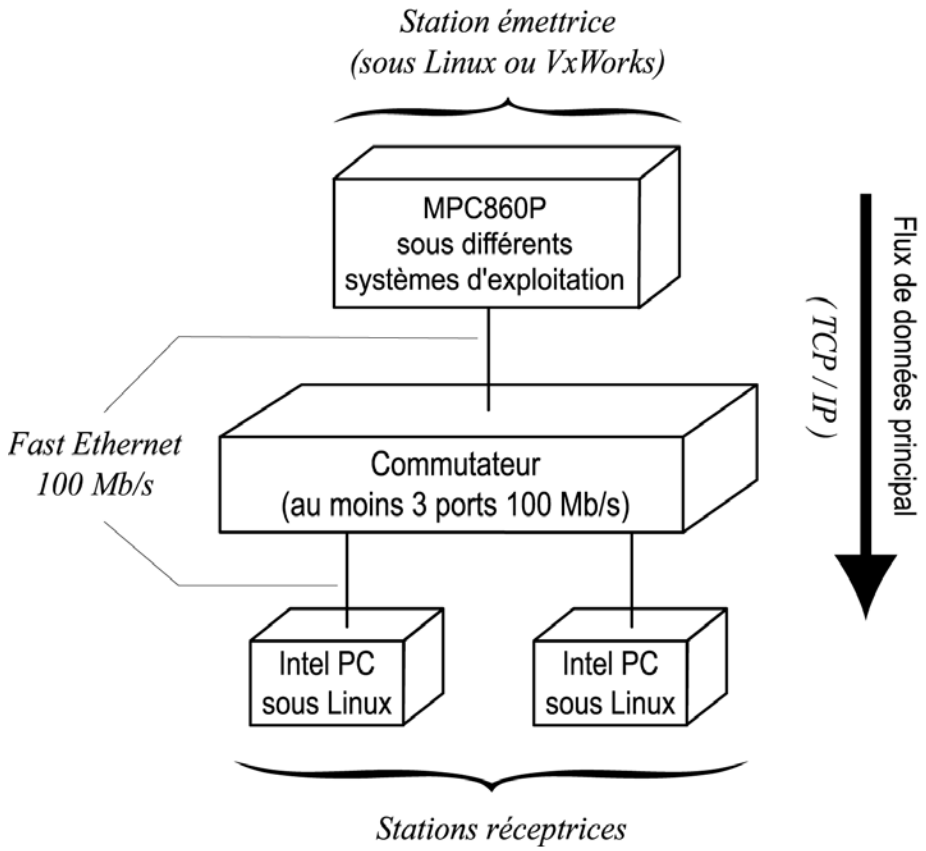


Fig. IV-20 : Dispositif prototype pour tests de débit sous différent systèmes d'exploitation

La recherche de la simplicité dans le développement nous conduit à préférer le protocole TCP/IP, d'autant que les *middlewares* comme ControlHost sont basés sur TCP/IP. De plus, la connaissance du comportement du système TDAQ nécessaire à l'analyse physique (Cf. II.A.4) exige que les éventuelles pertes de paquets soient connues et contrôlées ; en conséquence, l'usage éventuel de UDP nous imposerait de développer une gestion des paquets perdus. Bien entendu, si les tests montrent une grave insuffisance dans les débits obtenus, le passage à un protocole plus léger comme UDP voire à l'écriture directe de trames Ethernet, devra être envisagé.

Ainsi, ce premier exemple de dispositif prototype nous apportera des éléments quatitatifs concrets qui nous permettront d'effectuer des choix importants comme celui du système d'exploitation ou du protocole réseau. Le point essentiel dans la conception de tels dispositifs est de pouvoir isoler clairement les caractéristiques que l'on cherche à mesurer. Dans notre exemple, le choix de passer par un commutateur et deux stations réceptrices illustre bien ce souci puisqu'il permet de s'affranchir d'une éventuelle faiblesse dans la station réceptrice, qui fausserait les résultats.

IV.B.5.2 Valider des architectures et dimensionner le système

Par « valider une architecture », nous entendons ici la vérification qualitative et quantitative d'un algorithme d'acquisition ou de déclenchement en tenant compte de sa distribution sur le réseau. Cette validation s'accompagne nécessairement des conditions nécessaires au bon fonctionnement de l'architecture, notamment son dimensionnement.

Le dispositif le plus simple à partir duquel nous pouvons commencer à valider l'architecture de notre application est une simple station de travail sur laquelle les fonctionnements distribués sont émulés par des mécanismes multitâches. Une grande partie des asynchronismes sont ainsi testés et des interblocages* résolus. Les algorithmes d'assemblage d'événements et de déclenchement sont également déverminés jusqu'à un certain point. Bien entendu, tout ce qui est lié aux latences temporelles et aux bandes passantes est laissé de côté sur ce dispositif. C'est précisément pour cette raison qu'il vaut mieux commencer les tests par un dispositif monomachine, car cela permet justement de dissocier les problèmes de déverminage. De plus, si les développeurs savent dès le départ que le code qu'il écrivent doit pouvoir fonctionner sur une seule machine, ils sont psychologiquement contraints de concevoir l'application en séparant strictement ce qui relève de la communication et de qui relève de l'algorithmie proprement dite. Nous verrons plus en détail aux chapitres IV.B.7 et IV.C comment canaliser le travail de conception dans ce sens.

Les tests sur une seule machine permettent également d'estimer le dimensionnement du système complet. Par exemple, en y mesurant les performances d'un algorithme de déclenchement sur des événements complètement assemblés, nous pouvons évaluer approximativement le nombre N de machines nécessaires à un fonctionnement satisfaisant du système complet : $N = t_{algo} * \tau_{événements}$, où N est le nombre de machines nécessaires, t_{algo} est le temps moyen d'exécution de l'algorithme et $\tau_{événements}$ le taux moyen d'événements assemblés. Bien entendu une évaluation plus précise devra prendre en compte les fluctuations des temps de calcul et des taux d'événements. Dans la plupart des cas, seule une simulation complète ou, mieux, des tests sur un dispositif suffisamment réaliste fourniront *in fine*, les réponses précises nécessaires au dimensionnement complet du système.

Inversement, si l'estimation de ce nombre est trop grand pour être réaliste par

rapport aux possibilités techniques et budgétaires, le travail des développeurs consistera à modifier les algorithmes afin d'aboutir à une valeur plus raisonnable.

Au-delà du dispositif monomachine, il s'agit bien entendu de mettre en œuvre des dispositifs plus élaborés comme celui de la figure IV-21 afin de commencer à valider l'architecture du système TDAQ y compris dans ses aspects plus « bas niveau » liés aux communications sur réseau. Dans de tels dispositifs, le parallélisme intrinsèque doit être implémenté par au moins 2 nœuds sources de données afin d'effectuer au moins un assemblage d'événements sommaire dans le(s) nœud(s) récepteur(s). L'implémentation du parallélisme de performance par la mise en place de plus d'une machine réceptrice permettra de tester les algorithmes de gestion de la ferme de traitement. La complexification des dispositifs prototypes permet également d'introduire dans le système des temps de latence plus proches de ceux du système final et de détecter ainsi d'éventuelles erreurs de synchronisation qui seraient passées inaperçues sur le dispositif monomachine.

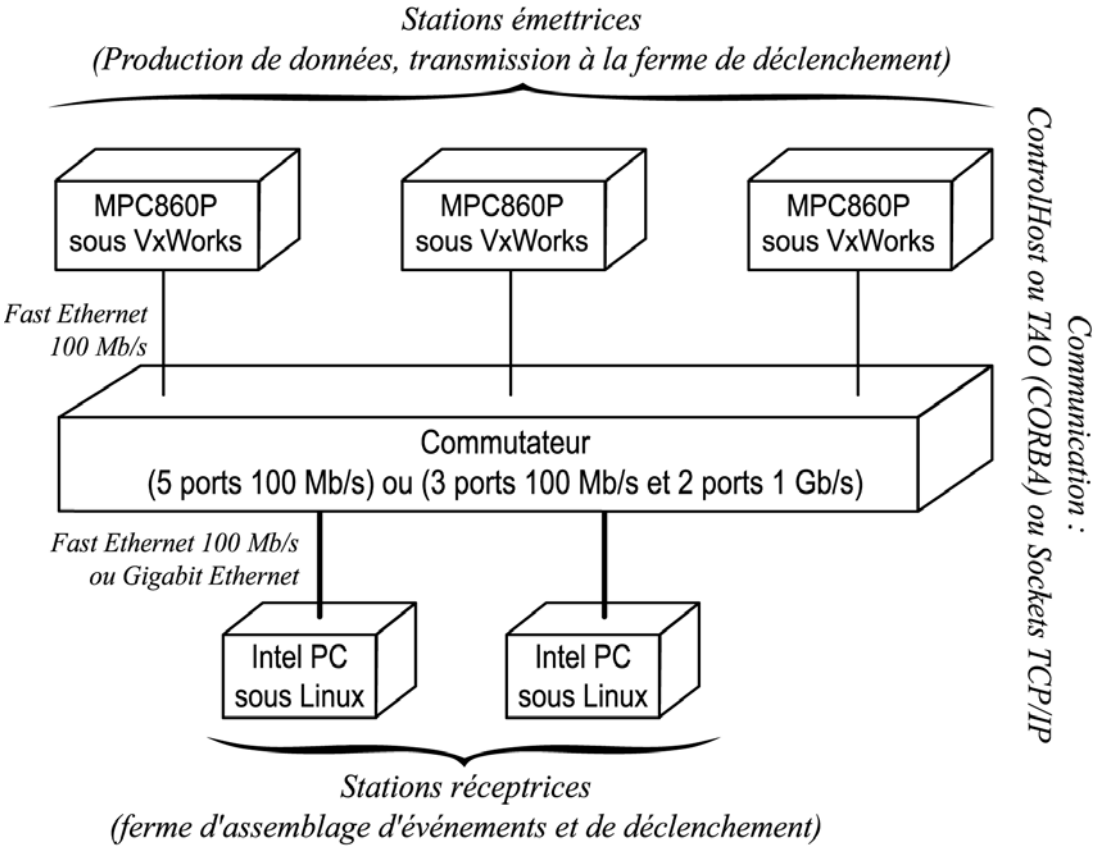


Fig. IV-21 : Dispositif prototype plus élaboré pour test d'architecture et d'algorithmie

IV.B.5.3 Tester des modules logiciels ou matériels développés

La production de dispositifs prototypes est également nécessaire pour avoir des

« bancs de test » permettant de valider le fonctionnement de certains modules développés pour l'application. En particulier, le test et le déverminage des objets implémentés sous forme matérielle ne peut en général s'effectuer sans un système d'acquisition adapté. Or, ces objets consistent justement en des modules destinés à s'intégrer dans un système TDAQ, ce qui implique qu'une partie de leur banc de test associé ressemble fortement au dispositif final auquel ils seront rattaché dans l'application réelle. Aussi, pourquoi ne pas s'appuyer sur les mêmes modules prototypes matériels et logiciels pour fabriquer lesdits bancs de tests. Le composant `Acq_MO_Amont` de notre application type inspirée d'ANTARES, par exemple, doit être implémenté sur FPGA. Il doit communiquer avec le composant `Acq_MO_Aval` implémenté sur le processeur MPC860P à travers une mémoire partagée. Par conséquent, tout dispositif prototype (logiciel compris) comme celui de la figure IV-21 peut également faire office de système d'acquisition dès lors que l'on remplace les cartes contenant un MPC860P par des cartes analogues intégrant en plus les FPGA envisagés (fig. IV-22).

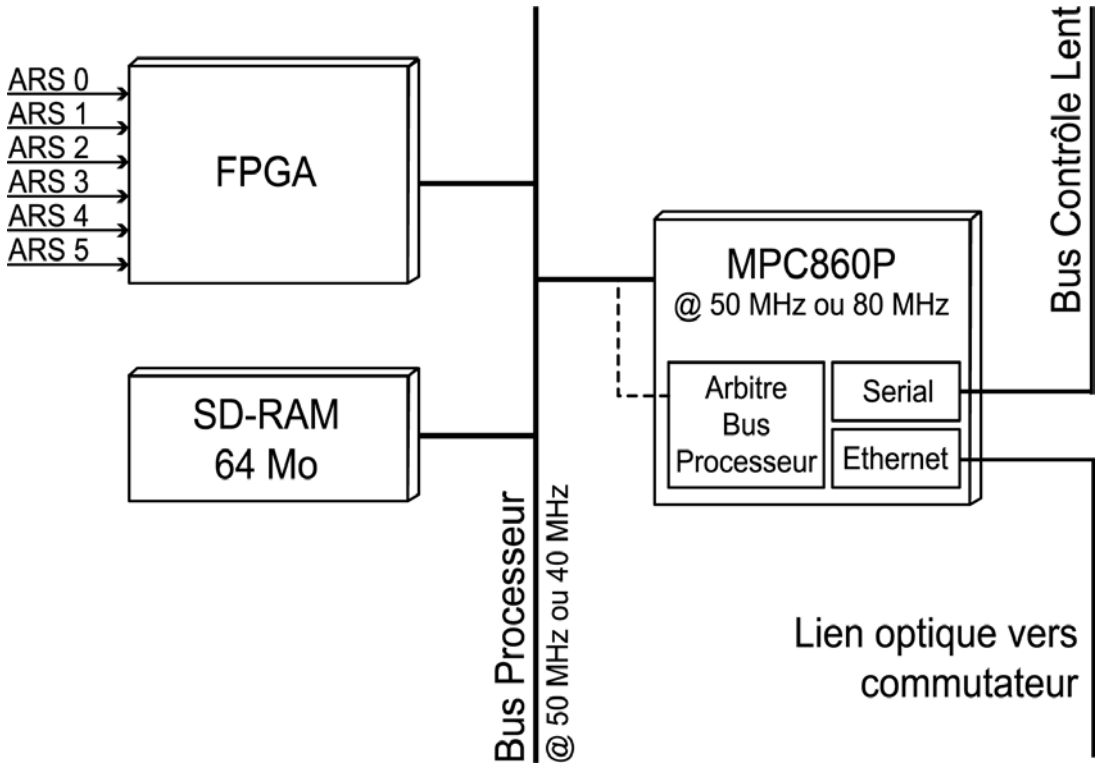


Fig. IV-22 : Diagramme blocs simplifié de la carte d'acquisition offshore d'ANTARES

Ainsi, l'architecture et surtout les composants logiciels développés pour les dipositifs prototypes peuvent directement servir à la fabrication de bancs de test pour certains modules spécialisés de l'application tels les modules sur FPGA.

IV.B.6 Tests et mesures

Cette activité de notre cycle de développement est bien entendu le corollaire naturel de l'activité de production de prototypes. Il s'agit en effet de procéder aux tests et aux mesures sur les prototypes produits afin de déverminer les modules matériels et logiciels associés et de mesurer les performances effectives afin d'effectuer des choix et d'orienter le travail pour la suite du développement.

L'activité « tests et mesures » n'étant possible qu'une fois les dispositifs prototypes idoines fabriqués, elle croît au cours du cycle de développement en même temps que l'activité « production de prototypes ». Mais, comme indiqué sur la figure IV-23, elle continue de croître même après que l'on ne fabrique plus de prototypes. En effet, la fabrication de prototypes tend à s'annuler à mesure que le(s) système(s) prototype s'approche(nt) du système TDAQ final, mais les tests et les mesures doivent continuer en s'intensifiant afin de mener encore plus loin la mise au point, le déverminage et la connaissance du système. On peut même dire que cette activité se poursuit durant toute la durée de vie de l'expérience à travers le monitorage et l'analyse continuelle du comportement du système.

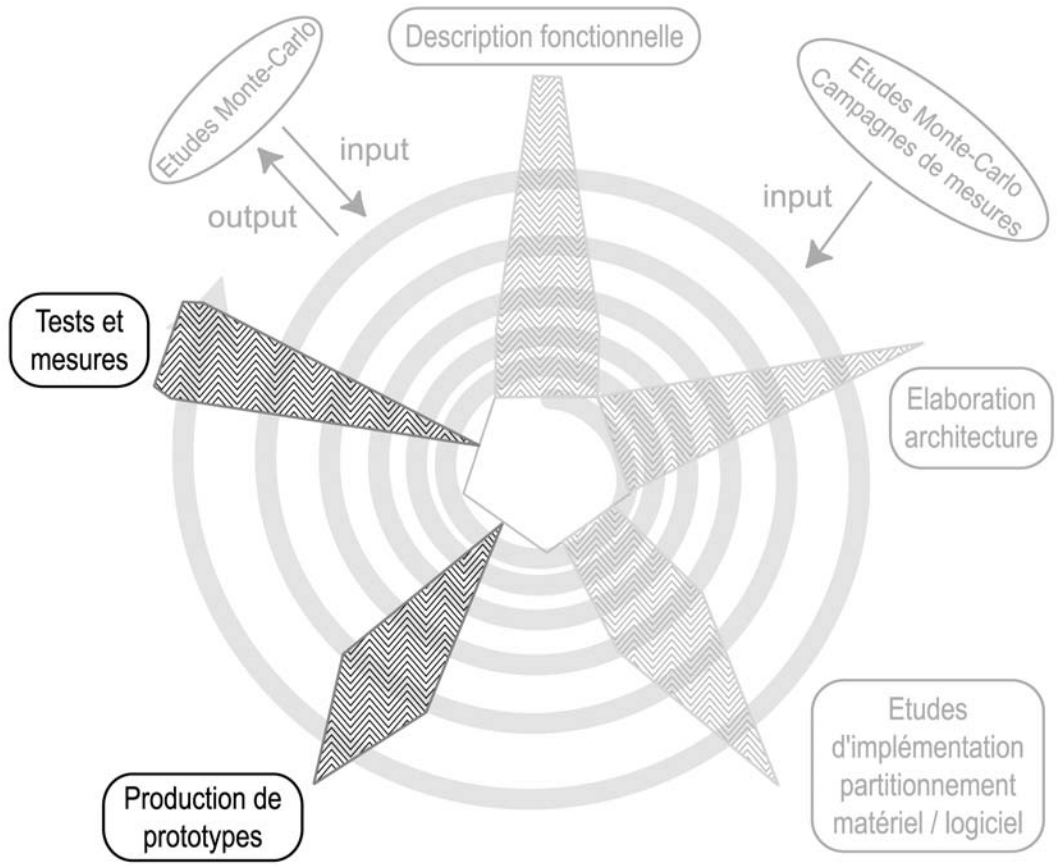


Fig. IV-23 : Phases de production de prototypes et de tests du cycle de développement TDAQ

Poursuivons, pour illustrer notre propos, notre exemple basé sur ANTARES. Le dispositif de la figure IV-20 nous permet donc de mesurer la bande passante utile d'émission de données depuis un processeur MPC860P sous Linux et VxWorks : celle-ci est respectivement de ~28 Mb/s et ~32 Mb/s en utilisant une pile IP standard. Cependant, le système d'exploitation VxWorks est doté d'un mécanisme spécifique de « *zero-copy buffers* » évitant toute recopie des paquets de données avant leur envoi. En utilisant ce mécanisme, le débit maximum transmis sous VxWorks avec TCP/IP atteint ~50 Mb/s. Ces tests sont clairement en faveur de VxWorks, et si nous retenons la contrainte des 40 Mb/s (incluant le facteur de sécurité), le choix de ce système d'exploitation s'impose.

D'autre part, le saut quantitatif de 32 Mb/s à 50 Mb/s obtenu uniquement par l'utilisation des *zero-copy buffers* indique que le goulot d'étranglement sur la plateforme MPC860P sera la bande passante mémoire, d'autant qu'elle devra être partagée avec le FPGA (fig. IV-22). Nous aurons donc intérêt dans tout notre développement à minimiser l'accès à la mémoire, notamment en réduisant la taille du code de façon à optimiser l'utilisation du cache d'instructions de 16 Ko.

Nous pouvons également obtenir des mesures utiles au dimensionnement du système sur un dispositif monomachine : une première mesure de l'algorithme de déclenchement sur une station Linux dotée d'un Pentium III à 1 GHz permet d'estimer la taille de la ferme de calcul pour le système TDAQ d'ANTARES à environ 100 machines (Cf. formule p. IV-45). Les calculs ont été menés sur un grand nombre de données réalistes produites par simulation Monte-Carlo. L'estimation, faite uniquement à partir de valeurs moyennes, est cependant assez précise car les calculs sont effectués sur des paquets représentant quelque 10 ms d'événements, laps de temps suffisant pour absorber les fluctuations sur des événements dont la durée de vie est de l'ordre de la microseconde (traversée du détecteur par une particule à la vitesse de la lumière). Il est donc clair que le sous-système d'aquisition et de déclenchement à terre sera basé sur une ferme de calcul comprenant un nombre important de stations. Il sera donc crucial de concevoir un algorithme de gestion de la ferme de calcul qui permettra aux nœuds d'acquisition offshore de distribuer correctement les données à la ferme tout en optimisant la bande passante du réseau et en minimisant les risques de congestion. En fait, il s'agira avant tout de concevoir une architecture logicielle qui nous permettra d'insérer facilement différentes politiques de gestion de ferme afin de déteminer

par des mesures sur prototypes (voire sur le système final) le meilleur algorithme.

Les résultats des tests et mesures orientent notre travail sur les activités d'élaboration d'architecture et d'implémentation durant les itérations suivantes du cycle de développement. Les exemples décrits illustrent clairement ce fait : le choix du système d'exploitation en faveur de VxWorks, les conclusions sur la bande passante mémoire qui vont nous conduire notamment à être très vigilants sur l'empreinte mémoire du code et donc, en particulier, des *middlewares*, et enfin, l'évolution des modèles statiques et dynamiques du système afin de tenir compte de la nécessité de mettre en œuvre la gestion de la ferme de calcul du sous-système de déclenchement.

La figure IV-24 représente ainsi une évolution possible du diagramme de séquence fonctionnel de la figure IV-6 : l'objet :`Acquisition_Globale` a été démultiplié (indexé sur la figure par la variable i), un objet de classe `GestFerme` a été inséré entre l'objet de classe `Transmission` et les objets de classe `Acquisition_Globale` et la requête d'acquisition `acq(fr_MO)` se décompose en une opération `destin(fr_MO)` qui détermine la destination du paquet de données `fr_MO` en renvoyant l'index `i` dans la ferme de calcul, suivie de l'appel `acq(fr_MO)` original à l'objet `acq_glob[i]`. La méthode `destin()` de la classe `GestFerme` implémente donc la *politique de gestion* de la ferme de calcul, c'est-à-dire l'algorithme permettant d'attribuer un élément de ferme à un paquet de données. On pourra ainsi mettre en œuvre différentes politiques de gestion de la ferme en spécifiant que `destin()` soit une méthode virtuelle et en créant autant de classes dérivées de `GestFerme` que l'on a de politiques différentes.

Remarquons qu'avant d'entamer l'itération suivante dans le cycle de développement, les résultats des mesures sont éventuellement utilisées pour affiner les programmes de simulation afin d'obtenir, d'une part, des estimations plus réalistes sur l'efficacité et le pouvoir de réjection du système TDAQ (Cf. VI.B) et, d'autre part, des évaluations plus précises sur les distributions statistiques des flux de données et sur le comportement global du réseau, notamment en ce qui concerne les risques de congestion.

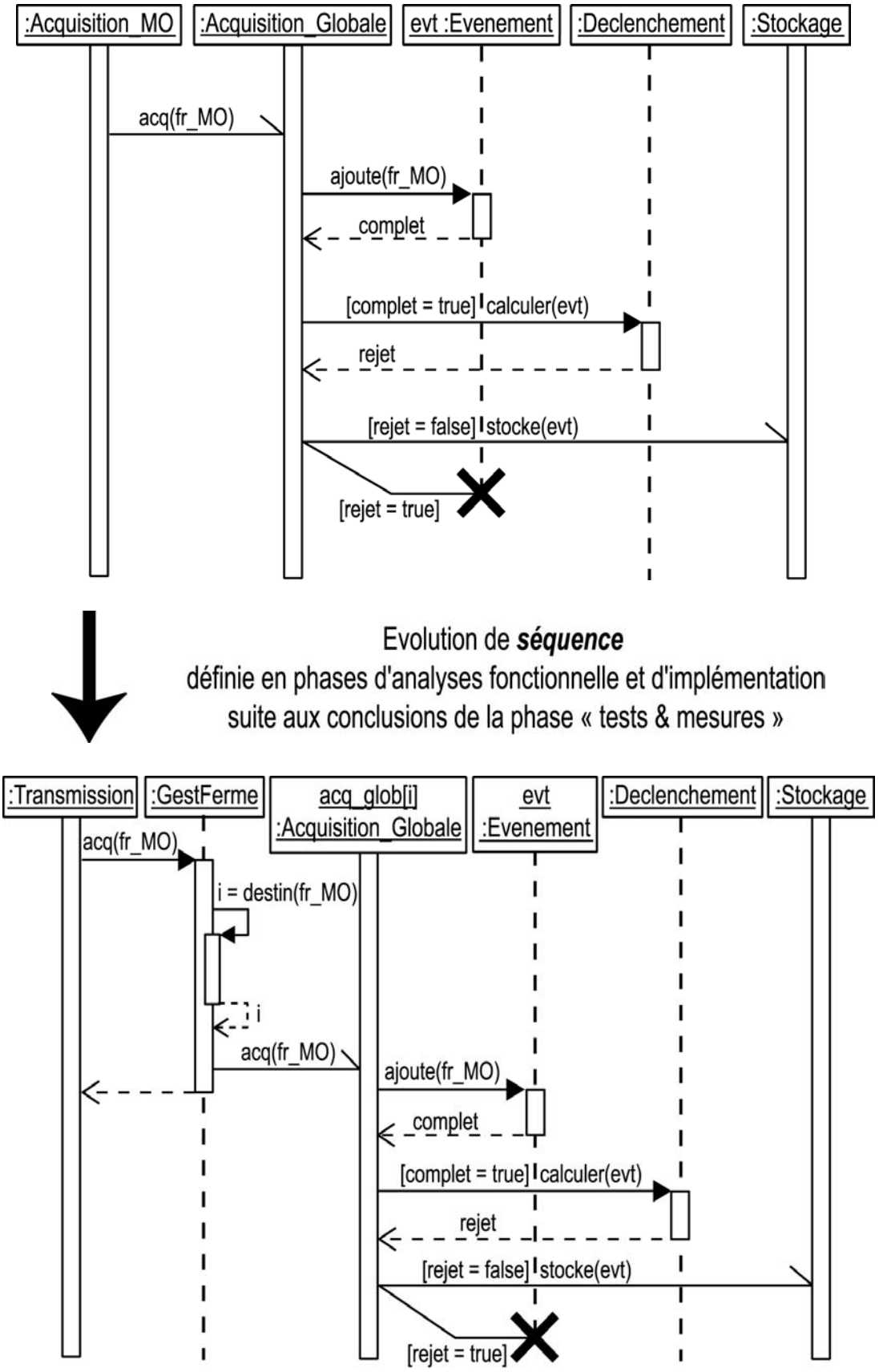


Fig. IV-24 : Evolution du modèle dynamique du système TDAQ d'ANTARES :
Séquence d'acquisition / déclenchement avec module de gestion de ferme de calcul

IV.B.7 Modélisation système et redéploiements

Lorsque l'on parle de modéliser un système, on a souvent tendance à ne garder à l'esprit que le modèle final. Or, l'activité de modélisation d'un système réel ne se limite pas à la production du modèle final. Bien entendu, c'est la mise en œuvre de celui-ci qui importe *in fine*, mais pour parvenir à cette étape, le processus de modélisation doit passer par la production et la spécification de nombreux modèles intermédiaires, éventuellement validés par des réalisations prototypes concrètes. Ce sont ces objets intermédiaires qui permettent, par une sorte de croissance « organique », l'élaboration effective du système réel. Nous avons ainsi vu au IV.B que notre cycle de développement des systèmes TDAQ était basé sur la prise en compte explicite de ces étapes intermédiaires puisqu'il se caractérise, entre autres, par un style de développement « incrémental » consistant à faire progressivement évoluer les modèles et les dispositifs prototypes par complexifications successives vers le système final.

L'un des points essentiels de notre propos est que la dynamique de cette croissance progressive du modèle provient des interactions successives entre le modèle fonctionnel du système (au sens défini au IV.B.2 p. IV-11) et ses multiples déploiements sur différentes plateformes et architectures d'implémentation. Nous voyons ainsi le processus de modélisation du système TDAQ comme une suite de *redéploiements* successifs du *même* système abstrait sur des plateformes de plus en plus complexes. Bien entendu, chaque redéploiement est susceptible de susciter ses propres développements ainsi que de rétroagir sur le modèle abstrait lui-même, mais il s'agira de distinguer clairement ce qui relève de modules nécessaires à un type d'implémentation particulier (par exemple, les modules assurant la communication des objets à distance) et les modules relevant de la spécification strictement fonctionnelle du système (c'est-à-dire du système abstrait). Le cycle de développement TDAQ décrit au IV.B.2 exhibe des exemples qui donnent un premier aperçu de cette dynamique provoquée par les redéploiements successifs du système selon de multiples schémas d'implémentation. En reprenant le diagramme de déploiement de la figure IV-15 (p. IV-27) et en le complétant à l'aide des nouvelles spécifications et des choix introduits par la suite lors des différentes phases illustrant le cycle de développement, nous obtenons le diagramme de la figure IV-25.

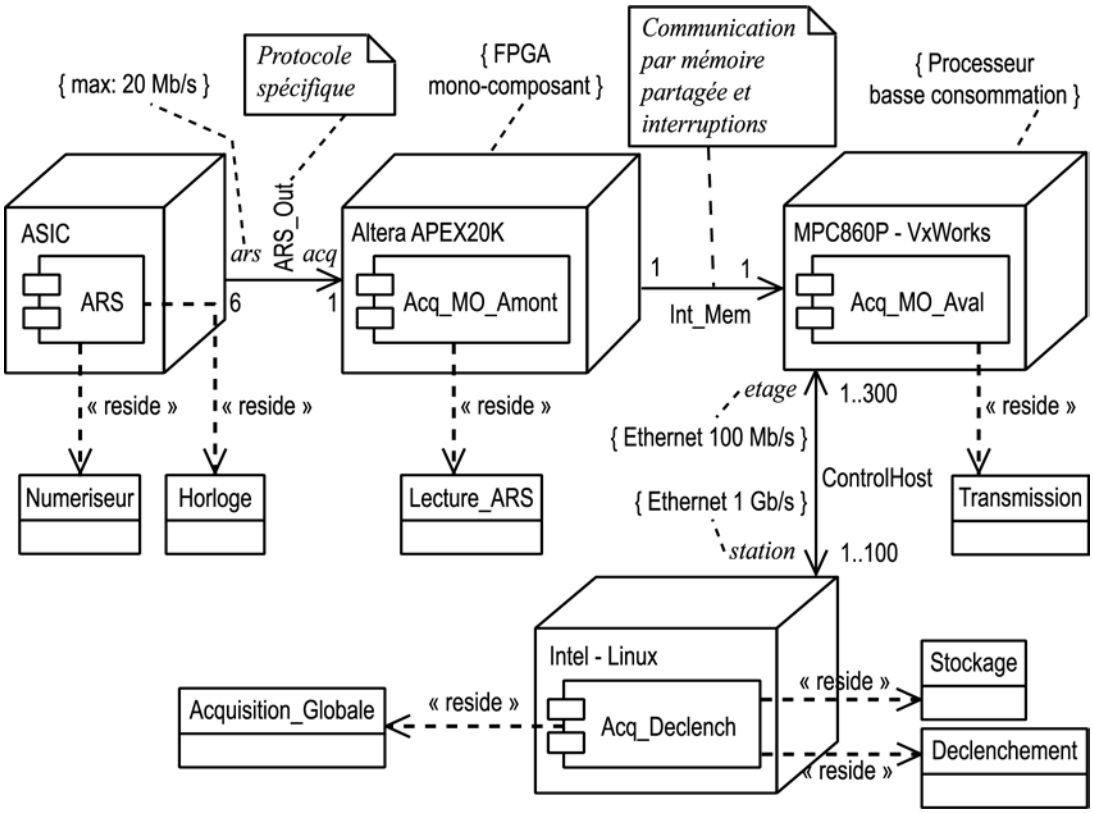


Fig. IV-25 : Evolution du modèle d'implémentation du système TDAQ d'ANTARES

Outre les spécifications plus précises concernant le choix des processeurs, des systèmes d'exploitation ou des technologies réseau, nous voyons également une évolution de la cardinalité du nœud `station` à terre : elle était de 1 à l'origine car fonctionnellement, nous n'avons pas besoin de plus d'une station d'assemblage d'événements et de déclenchement. Elle est passé à un nombre entre 1 et 100 car nous avons vu que seule une *ferme* de calcul pourrait traiter les flux de données attendus et les calculs nécessaires. En quoi ce modèle de déploiement provoque-t-il le développement de modules spécifiques et fait-il évoluer le modèle abstrait ?

Nous avons par exemple vu que la multiplication de la cardinalité associée au rôle `station` nécessite l'introduction d'une classe liée à la gestion de la ferme de calcul. Aussi, de même que nous avons du procéder à l'évolution de séquence de la figure IV-24, devons-nous faire évoluer le diagramme de classe correspondant comme indiqué sur la figure IV-26.

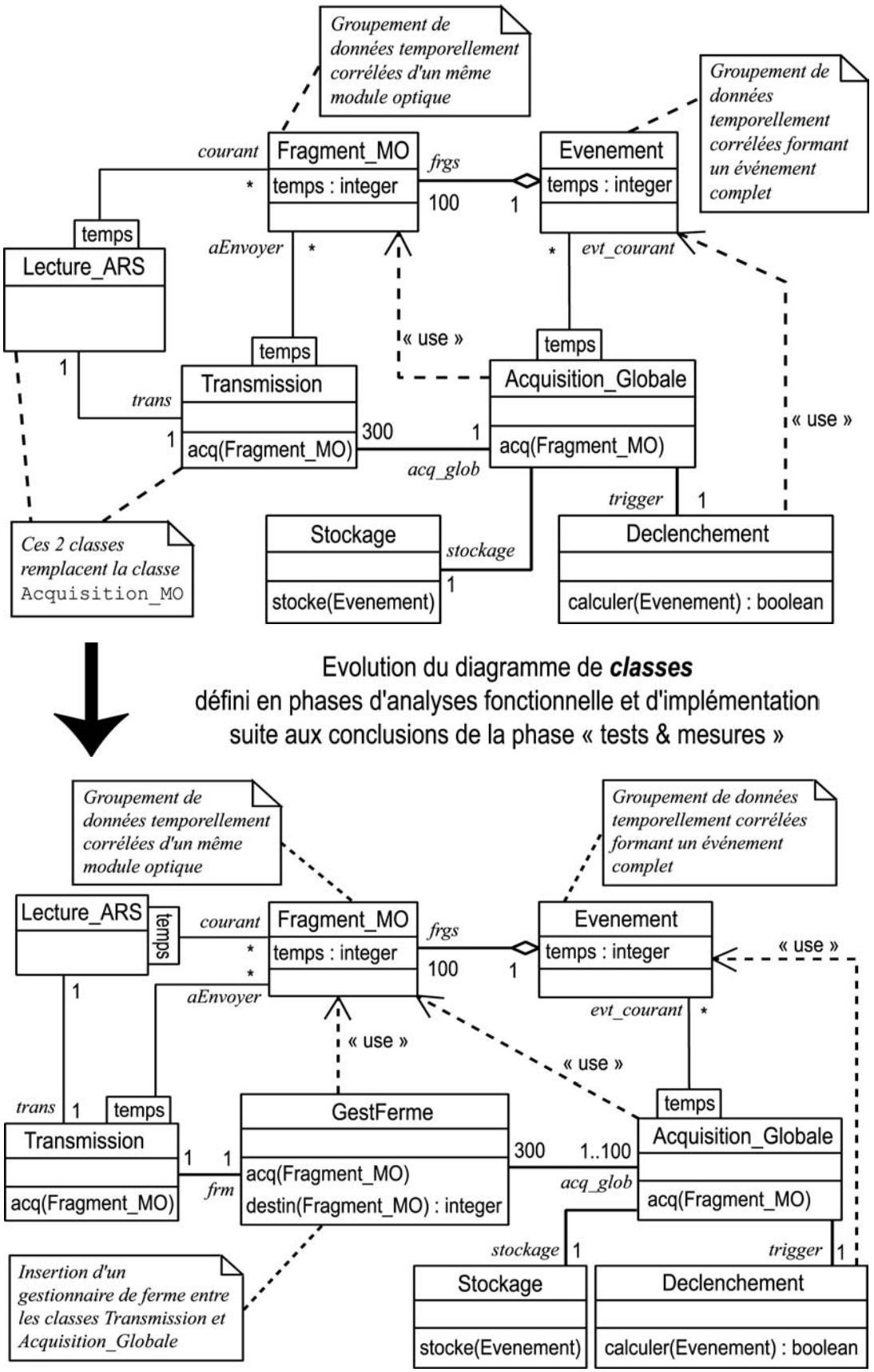


Fig. IV-26 : Deuxième évolution des classes de l'acquisition globale

D'autre part, nous n'avons pas précisé jusqu'ici selon quelles modalités devaient s'effectuer les appels de méthodes en dehors d'un espace d'adressage donné. Nous nous sommes juste contentés d'associer des protocoles ou des *middlewares* de communication aux associations entre nœuds sur les diagrammes de déploiement, spécifiant par là que la communication entre objets de nœuds différents sera basée sur tel protocole ou tel *middleware*. Par exemple, le diagramme de la figure IV-25 montre que la communication entre les objets du nœud `MPC860P_VxWorks` et ceux du nœud `Intel_Linux` est assurée par `ControlHost`. Il s'agira donc de spécifier également selon quelles consignes de modélisation ces mêmes objets devront s'interfacer avec les modules de communication inter-nœuds. Ainsi, à chaque fois que l'on spécifie un protocole ou un module particulier pour mettre en œuvre la communication entre deux types de nœuds, on doit également fournir *in fine* une description non ambiguë de la manière dont un appel de méthode doit être interprété dans le cadre de ce protocole. Cette description peut être assimilée à un motif récurrent*, ou plutôt à une spécification programmatique (non ambiguë) d'un motif récurrent*. Dans les sections IV.C et au-delà, nous verrons comment fournir cette description sous forme de procédures de transformation de modèle dans le cadre d'un AGL* UML*.

Etant donné que les langages de programmation auxquels nous avons aujourd'hui affaire dans le cadre des développements de systèmes TDAQ[14] ne comportent pas la notion d'« appel asynchrone », il appartient également aux développeurs de fournir des consignes précises sur la manière de mettre en œuvre concrètement ces appels. Par conséquent, pour chaque type de nœud comportant des appels asynchrones, les développeurs doivent fournir (ou utiliser) un module logiciel matérialisant les motifs récurrents* selon lesquels les appels asynchrones doivent être mis en œuvre.

Nous voyons donc que la spécification d'un modèle de déploiement doit nécessairement s'accompagner de spécifications et de développements supplémentaires sans lesquels la mise en œuvre effective et concrète du modèle fonctionnel sur le déploiement en question serait impossible. Le modèle *réel*, c'est-à-dire celui qui est finalement compilé et instancié sur le système résulte d'une *transformation* du modèle fonctionnel selon le modèle de déploiement et ses motifs récurrents

14. Essentiellement C, C++ et Java pour le logiciel et VHDL pour le firmware* *et les ASIC**.

associés. Et cette transformation doit être réitérée pour chaque classe de déploiement différent : le modèle réel obtenu à partir du modèle fonctionnel ne sera pas le même selon que l'on veut produire une application monomachine, un prototype de test ou le système TDAQ au complet dans son déploiement final. Pour nous résumer, nous pouvons dire que la dynamique incrémentale de la modélisation de notre système TDAQ s'articule donc autour du triptyque [Modèle fonctionnel – Modèle de déploiement – Motifs récurrents]. Nous verrons plus loin qu'il nous sera également nécessaire d'y ajouter un modèle d'instanciation. Pour l'instant, retenons simplement que le processus de développement d'un système TDAQ se caractérise par les points suivants :

- un modèle fonctionnel que l'on étoffe à chaque itération du cycle
- une série de redéploiements mettant en œuvre le modèle fonctionnel
- chaque redéploiement correspond un modèle de déploiement associé à des motifs récurrents d'implémentation
- les redéploiements tendent vers le déploiement final par complexification progressive.

IV.C FORMALISATION DU PROCESSUS DE MODÉLISATION

Dans cette sous-section, nous nous attacherons à spécifier une structure de conception adaptée à notre cycle de développement TDAQ et à sa philosophie basée sur les redéploiements d'un même système abstrait. Notre langage principal sera UML, éventuellement complété par des extensions ou restrictions spécifiques.

IV.C.1 Cycle de développement et modélisation

Notre structure de conception reposera sur une arborescence de paquetages décrite au IV.C.2, chaque type de paquetage étant dévolu à une classe d'activités de conception bien précise. Notre ligne de conduite lors de l'élaboration de cette structure de conception devra être guidée par le souci constant de mettre en œuvre des procédures de travail et des motifs récurrents* automatisables et ayant pour objectif la génération de code (semi-)automatique. C'est cette structure, mise en œuvre dans un AGL*, qui doit *in fine* aboutir à notre canevas TDAQ annoncé en III.C.5.

Outre un découpage en paquetages dévolus à des spécifications, notre environ-

nement de développement nécessite que l'on définisse un sens de progression dans le processus de développement d'un système TDAQ. Cette orientation doit bien entendu s'inscrire dans le cycle de développement exposé au IV.B et refléter le principe, exposé au IV.B.7, de développement « organique » du système par redéploiements multiples et évolution d'un même système abstrait. Pour cela, nous décrirons au IV.C.4, en les justifiant, les différentes procédures associées à notre canevas de développement ainsi que notre usage des diagrammes et de la sémantique UML au cours de chacune des phases de développement ainsi que leur interactions mututelles.

Nous utiliserons bien entendu plusieurs types de diagrammes UML dans des buts bien précis, mais nous ne ferons pas usage de la totalité des possibilités offertes par UML en la matière. En particulier, les diagrammes de cas d'utilisation ne nous serons, en pratique, d'aucune utilité et ce, pour des raisons spécifiques exposées au IV.C.3.3. Nous exposerons également au cours de notre description les annotations particulières et extensions UML spécifiques dont nous aurons besoin pour baliser le travail de développement TDAQ. Afin de fixer les idées et de clarifier notre propos sans nous encombrer de complexités inutiles, nous recourrons à des exemples isolés, très simples en soi, mais dont la généralisation à des cas réalistes sera ou bien évidente, ou bien dûment justifiée.

Certains produits de la conception relèveront de l'activité du développeur de systèmes TDAQ, c'est-à-dire de l'ingénieur en électronique / informatique en charge du développement d'un sous-système TDAQ particulier. Nous désignerons les individus rattachés à cette classe d'activités par les terme de « développeur » ou de « concepteur » et le produit de leur travail par les termes « module » ou « produit ». Ces produits sont ceux qui doivent faire l'objet d'un développement à chaque nouveau projet.

D'autres produits de la structure de conception seront le fait du développeur de canevas, c'est-à-dire qu'ils correspondent à des modules utiles à la structure de conception et, notamment, à son automatisation. Quand il y aura un risque d'ambiguïté, nous désignerons les individus rattachés à cette classe d'activités par les terme de « méta-développeur » ou de « méta-concepteur » et le produit de leur travail par les termes « méta-module » ou « méta-produit », sinon les termes simple seront employés lorsque le contexte l'autorisera. Les méta-modules seront bien

entendu utiles à tout développement de système TDAQ et ne seront pas spécifiques à système particulier.

IV.C.2 Organisation des paquetages

Chaque nouveau développement d'un système TDAQ correspond à un nouveau projet. Appelons `ProjetTDAQ` le paquetage racine contenant la totalité des éléments de modélisation et de spécification du projet, y compris les éventuels paquetages importés, ainsi que tout le code développé. La première exigence que l'organisation de notre structure de modélisation doit satisfaire est l'explicitation du triptyque [Modèle fonctionnel – Modèle de déploiement – Motifs récurrents] exposé au IV.B.7 (introduit p. IV-56). Le paquetage `ProjetTDAQ` devra donc se scinder en 3 sous-paquetages : celui contenant le modèle fonctionnel, appelé `Fonctionnel`, celui contenant les modèles de déploiement, appelé `Deploiements` et celui contenant les motifs récurrents utilisés, appelé `Generiques`. L'appellation `Generique` se justifie par le caractère indépendant des modules concernés par rapport à toute application particulière, mais surtout par le fait que ces modules définiront essentiellement des procédures de transformation automatique de modèle et se situeront ainsi au niveau du méta-modèle. Ce dernier paquetage ne devrait donc contenir, en toute logique, que des modules (ou plutôt des méta-modules) importés. En réalité, il contiendra éventuellement des modules, certes génériques –et donc réutilisables pour d'autres projets– mais développés *à l'occasion* du développement de ce projet particulier. En effet, rappelons la conclusion de notre argumentation du III.C.4, à savoir que le développement de modules génériques réellement utilisables ne peut se faire qu'à partir d'applications réelles.

Nous avons vu que le processus de modélisation équivaut à des redéploiements successifs d'un même modèle abstrait qualifié de « fonctionnel ». Si nous mettons de côté les questions de versionnement*, nous pouvons donc dire qu'il n'y a qu'un seul modèle fonctionnel du système TDAQ alors qu'il n'y a *a priori* aucune limite au nombre de modèles de déploiement. Quant aux motifs récurrents* implémentés dans les méta-modules, il sont bien entendu en nombre quelconque. Comme indiqué sur la figure IV-27, Les paquetages `Deploiements` et `Generiques` contiendront donc eux-mêmes un nombre quelconque de sous-paquetages, chacun correspondant respectivement à un modèle de déploiement ou un motif

récurrent particulier.

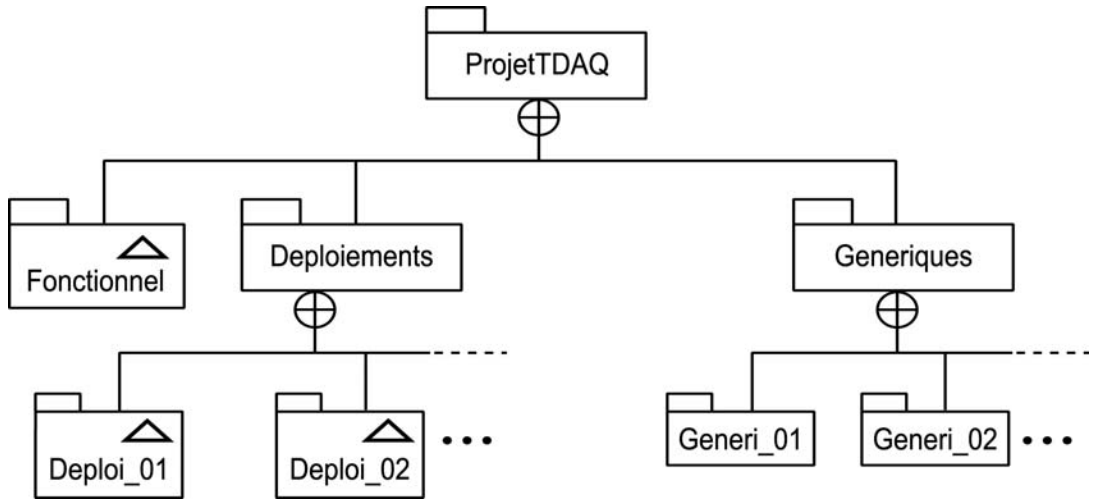


Fig. IV-27 : Arborescence des paquetages d'un projet de développement TDAQ

Enfin, les paquetages ainsi définis sont non seulement destinés à contenir les éléments des *modèles* (tels les classes) de notre projet mais également à en organiser les éléments de *modélisation* tels les différents diagrammes. Par exemple, les diagrammes définissant le déploiement « `MonoMachine_Linux` » seront associés au paquetage :

`ProjetTDAQ::Deploiements::MonoMachine_Linux`.

IV.C.2.1 Paquetage « Fonctionnel »

Le paquetage `Fonctionnel` contient le modèle complet du système TDAQ envisagé selon le point de vue strictement « fonctionnel » au sens du IV.B.2[15]. En d'autres termes, ce que l'on spécifie au niveau de ce paquetage doit être valide quel que soit le déploiement choisi, et donc, indépendant des problématiques d'implémentation. Par exemple, une cardinalité plurielle ou un « gestionnaire de ferme » introduits pour mettre en œuvre un parallélisme de performance ne doivent pas apparaître au niveau des modèles fonctionnels. D'une manière générale, toutes classes, tous paramètres et toutes contraintes liés à la *mise en œuvre explicite* des problématiques de distribution et de concurrence n'ont pas leur place dans les modèles du paquetage `Fonctionnel`.

En se référant à l'exemple d'ANTARES présenté dans les sous-sections IV.B et IV.B.7, nous pouvons dire que les diagrammes de séquence et de classes des

15. Comme indiqué par l'icône triangulaire fig. IV-27, c'est un paquetage de type « modèle » tel que défini par UML dans [B-12], pp. 2-188 à 2-189, 3-24 à 3-25.

figures IV-5, IV-6, IV-8, IV-9, IV-13, IV-14 (ainsi que les éléments représentés) appartiennent, pour l'essentiel, au paquetage `Fonctionnel`, avec une réserve en ce qui concerne le deuxième diagramme de la figure IV-13 en raison de la cardinalité 6 sur l'association entre les classes `Numeriseur` et `Lecture_ARS` (côté `Numeriseur`). En effet, *fonctionnellement*, nous n'avons que trois photomultiplicateurs par étage et n'avons donc besoin dans l'absolu que de 3 numériseurs. Le doublement de ces derniers est dû à un problème de performances puisqu'il est destiné à réduire le temps mort de numérisation. Cependant, ce composant étant nécessairement matériel (car traitant des signaux analogiques en entrée), nous pourrions considérer l'objet de classe `Numeriseur` comme une paire de composants ARS et l'appartenance du deuxième diagramme de classe de la figure IV-13 au paquetage `Fonctionnel` serait alors complètement acquise.

En revanche, les diagrammes des figures IV-24 à IV-26 ne sont clairement pas de type fonctionnel puisqu'ils intègrent l'implémentation de l'objet `:Acquisition_Globale` sur une ferme de calcul, ce qui est, bien entendu, spécifique à un déploiement particulier introduisant du parallélisme de performance sur l'assemblage d'événements et le déclenchement.

Afin de mieux caractériser le paquetage `Fonctionnel`, considérons le cas particulier des classes de ce paquetage. On peut dire qu'une classe appartenant au paquetage `Fonctionnel` ressemble à une classe-type (classe affectée du stéréotype «`type`») au sens défini par la spécification 1.3 d'UML ([B-12] pp. 3-49 et 3-50), c'est-à-dire une classe participant à la spécification d'un comportement, mais destinée à être *réalisée* par une classe d'implémentation que nous spécifierions dans un paquetage de déploiement. Cependant, UML impose explicitement qu'une telle classe ne peut avoir *que* des opérations, à l'exclusion de toute méthode[16], rapprochant ainsi sa sémantique de celle de l'« interface » ([B-11] p. 3-48). Or nous n'excluons pas *a priori* que le corps de certaines méthodes soit déjà défini au niveau fonctionnel. Par conséquent, la réalisation d'une classe-type par une classe d'implémentation au sens d'UML est une notion trop restreinte pour nos besoins et malgré une certaine proximité sémantique des classes du paquetage `Fonctionnel` d'avec les classes-types d'UML, nous ne les assimilerons pas à

16. UML distingue l'« opération » de la « méthode », la première n'étant qu'une signature de méthode et la seconde étant une implémentation de la première, dotée d'un « corps » d'instructions.

celles-ci. En revanche, nous pouvons maintenir une forme de relation de « réalisation » entre les classes des différents paquetages de déploiement et les classes correspondantes dans le paquetage fonctionnel.

IV.C.2.2 Paquetage « Deploiements »

Dans le formalisme d'UML, un déploiement est essentiellement composé de nœuds, comprenant des composants, et d'associations entre ces nœuds. Les associations représentent des canaux de communication ([B-12], p. 3-173).

Le paquetage `Deploiements` ne représente, au premier abord, que le regroupement de tous les sous-paquetages correspondant aux multiples déploiements envisagés durant le développement. En fait, il va également comprendre tout les éléments de modélisation communs aux différents déploiements. En effet, deux déploiements distincts sont tout-à-fait susceptibles de se référer, par exemple, à un même type de nœud de calcul, auquel cas les spécifications de ce nœud doivent être factorisées directement au niveau du paquetage `Deploiements` (Cf. fig. IV-28). Ces nœuds représentent une « plateforme » sur laquelle différents composants de l'application sont susceptibles d'être déployés. Les spécifications de cette plateforme peuvent, par exemple, comprendre le type de processeur, le système d'exploitation ainsi que les compilateurs et autres outils de production associés. Ainsi, lors de la spécification d'un déploiement particulier, le développeur a à sa disposition un « menu » de types de nœud qui représentent autant de plateformes d'implémentation différentes. Il pourra, bien entendu, enrichir ce menu en y rajoutant ses propres spécifications de nœud. Un 'Nœud' (*Node*) au sens d'UML est un 'Classifieur' (*Classifier*), ce qui en fait également un 'Espace de nommage' (*Namespace*) (Cf. [B-12], pp. 2-16 et 2-13)[17]. Aussi, tous les éléments de modélisation qui concourent à la spécification d'un nœud pourront-ils être inclus dans l'espace de nommage de ce nœud (le nœud joue alors le rôle d'un paquetage pour ses propres éléments de spécification).

Il nous reste à expliciter l'usage des sous-paquetages du paquetage

17. Remarque : la spécification UML peut prêter à confusion en ce qui concerne la sémantique de la métaclasse Node*. En effet, le chapitre sur la sémantique d'UML, auquel nous avons choisi de nous référer ici, définit bien* Node *comme dérivant de la métaclasse* Classifier*, mais le chapitre sur la notation UML ([B-12] pp. 3-172 à 3-176) semble, par abus de langage, utiliser le terme de* « Node » *pour désigner des instances et invoque la notion de « type de nœud » sous le vocable* « Node-Type » *pour signifier le classifieur. Cette confusion n'est bien entendu pas problématique tant que le contexte interdit toute ambiguïté.*

`Deploiements`. Ceux-ci sont naturellement consacrés chacun à la spécification d'un déploiement particulier du système fonctionnellement défini dans le paquetage `Fonctionnel`. Chacun d'eux délimite un *modèle de déploiement complet* du système TDAQ, ce qui justifie qu'il soit représenté par la spécialisation « Model » de la métaclasse « Package », dénotée par l'icône triangulaire sur les figures IV-27, IV-28 et IV-29 (Cf. [B-12], pp. 2-188 à 2-189, 3-24 à 3-25). Ils contiennent les éléments introduits par les diagrammes attachés à ce déploiement particulier. Ces diagrammes comprendront avant tout des nœuds sur lesquels les composants de l'application seront mis en œuvre. Chaque type de nœud, outre qu'il représentera une plateforme d'exécution particulière, remplira au sein de l'application, un rôle particulier. Aussi, les nœuds définis dans le cadre d'un déploiement particulier (un sous-paquetage de `Deploiements`) devront-ils être des spécialisations des nœuds généraux du paquetage `Deploiement` (Cf. fig. IV-28).

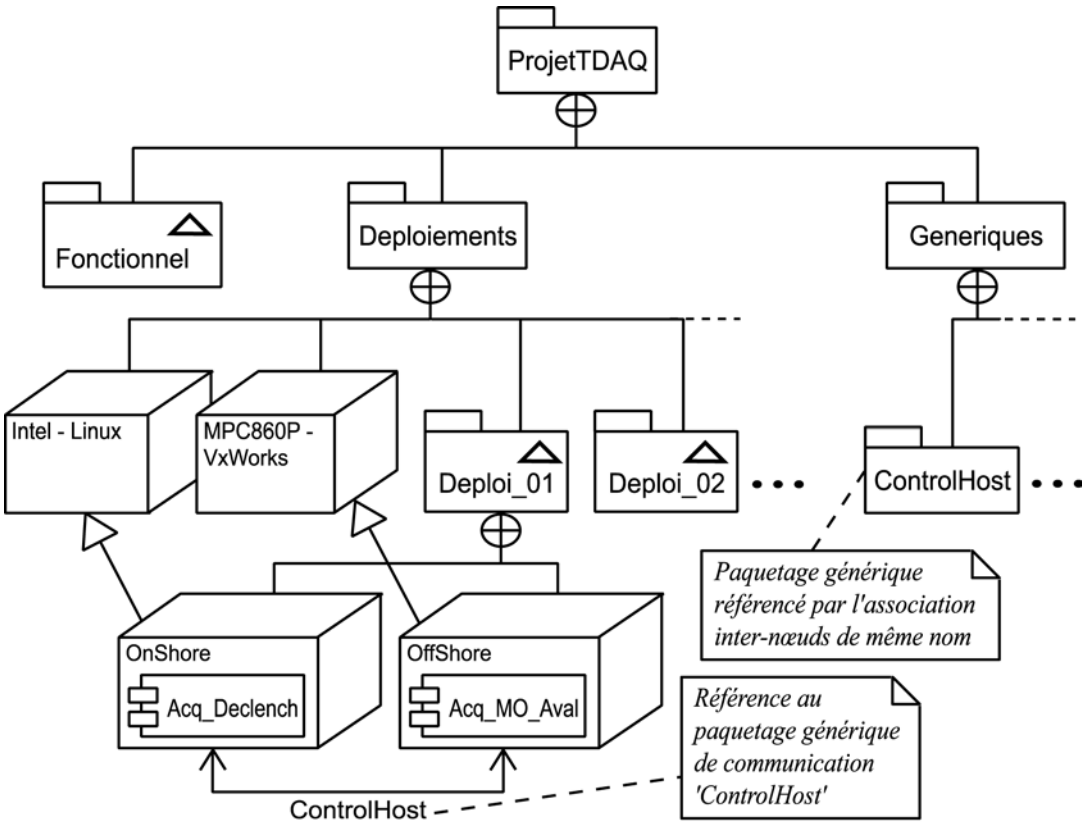


Fig. IV-28 : Associations inter-nœuds du paquetage `Deploiements` et méta-modules du paquetage `Generiques`

Un point que nous allons détailler au IV.C.4 et qui constitue l'une des caractéristiques importantes de notre structure de conception est le lien que nous établissons entre les associations de nœuds et les motifs récurrents liés à la distribution. Ainsi,

chaque sous-paquetage du paquetage `Deploiements` est destiné à recevoir également les associations de communication inter-nœuds définis par le développeur pour son déploiement particulier, associations dont le nom doit correspondre au nom d'un sous-paquetage du paquetage `Generiques` mettant en œuvre les modalités du protocole de communication en question. La figure IV-28 illustre par exemple un contenu de deux nœuds (`OnShore` et `OffShore`) associés par un mode de communication (`ControlHost`) tel que spécifié dans le diagramme de déploiement de la figure IV-25 (p. IV-53) où l'on remplacerait les nœuds `MPC860P-VxWorks` et `Intel-Linux` par les nœuds qui en dérivent sur la figure IV-28. Entre ces deux nœuds, l'association `ControlHost` se réfère à un méta-module du paquetage `Generiques` implémentant la communication inter-composants à travers le *middleware* ControlHost. Par exemple, si un objet de classe `Transmission` appartenant au composant `Acq_MO_Aval (`fig. IV-25 et IV-28) sur une instance de nœud `OffShore` dérivant de `MPC860P-VxWorks`, (fig. IV-28) a besoin de communiquer avec un objet de classe `Aquisition_Globale` appartenant au composant `Acq_Declench (`fig. IV-25 et IV-28) tournant sur une instance de nœud `OnShore` dérivant de `Intel-Linux` (fig. IV-28), alors cette communication doit se faire selon le motif récurrent défini par le lien `ControlHost`.

La spécification d'un déploiement particulier nous indique l'ensemble des espaces d'adresse utilisés ainsi que tous les composants y résidant. Ainsi, à partir d'un déploiement complètement spécifié, nous pouvons procéder :

- à la transformation du modèle fonctionnel selon les motifs récurrents référencés par le déploiement afin d'obtenir un modèle compilable.
- à la création d'un sous-système[18] (et donc d'un sous-espace de nommage additionnel) pour chaque espace d'adresse spécifié sur le déploiement, contenant les classes effectives qui lui sont attachées.

La figure IV-29 nous montre les sous-systèmes `OffShore_Subsys` et `OnShore_Subsys` produits à partir des nœuds `OffShore` et `OnShore`, chacun définissant un espace d'adresse séparé. Outre les classes homonymes de celles

18. Spécialisation « Subsystem » de la métaclasse « Package » (Cf. [B-12], pp. 2-188, 2-192, 3-19 à 3-22). Ce choix est justifié du fait que ce paquetage regroupe tout le code du système destiné à tourner effectivement sur le nœud homonyme. De plus, c'est un sous-système instantiable car il peut donner lieu à des instantiations multiples sur autant de nœuds de même type.

provenant du modèle fonctionnel, les entités incluses dans ces sous-paquetages devront bien entendu comprendre les classes produites par la transformation du modèle fonctionnel selon les motifs récurrents spécifiés par le déploiement (`ControlHost` dans l'exemple).

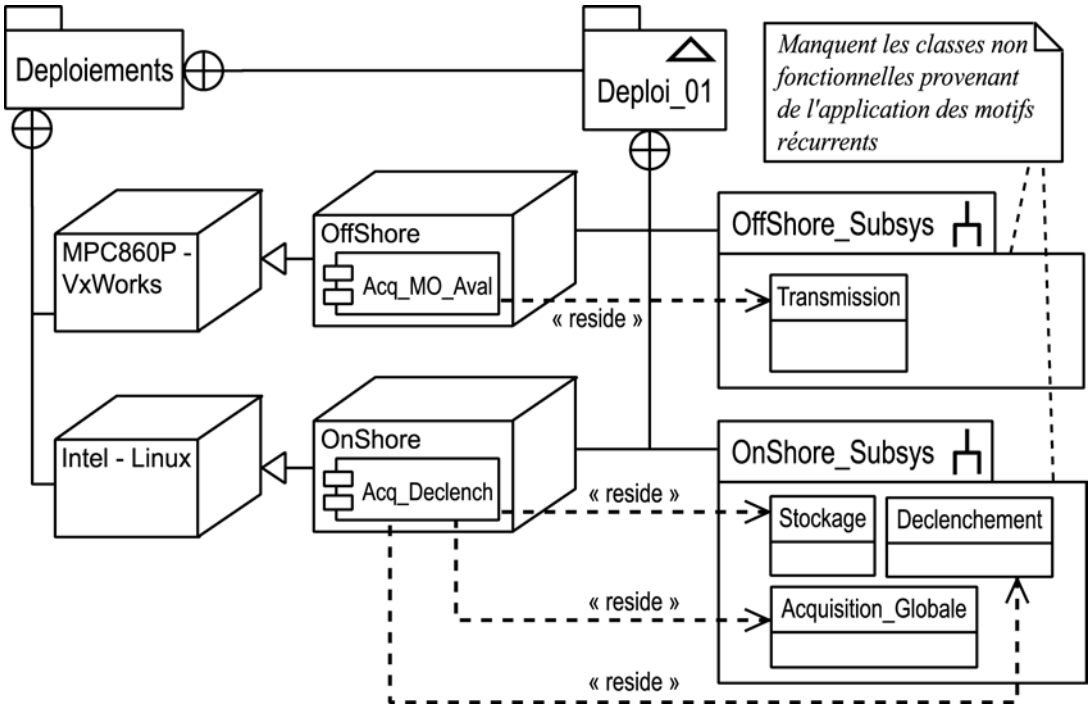


Fig. IV-29 : Sous-paquetages de classes concrètes associés à chaque espace d'adresse

IV.C.2.3 Paquetage « Generiques »

Les sous-paquetages du paquetage `Generiques` vont regrouper l'ensemble des spécifications et procédures liées à des motifs récurrents particuliers. Ils vont ainsi correspondre à ce que nous avons appelé des « méta-module » logiciels. Le cœur d'un tel méta-module sera constitué des procédures s'appliquant à des métaclasses pour effectuer des *transformations de modèle*. Notre approche des problématiques liées au développement des systèmes TDAQ va privilégier l'implémentation de méta-modules liés à la distribution et à la concurrence, puisque nous établissons un lien entre le nommage des associations inter-nœuds des modèles de déploiement et ces méta-modules. Mais il est clair que cette architecture est extensible à toutes autres notions susceptibles d'être exprimées dans un modèle de déploiement.

Un concept comme l'appel asynchrone, par exemple, exprimable sur un diagramme de séquence UML, ne possède pas d'interprétation univoque dans les langages de programmation orientés objet les plus répandus. Aussi devons-nous

développer des motifs récurrents permettant d'en donner une interprétation directement transposable en C++ ou en Java. Plus précisément, nous devrons diposer de méta-modules (au moins un) liés à la l'interprétation des appels asynchrones et dont la fonction consistera essentiellement à transformer les diagrammes de séquence comportant des appels asynchrones en diagrammes équivalents ne comportant que des appels synchrones sur des objets mettant en œuvre l'asynchronisme (nous en verrons un exemple plus loin). L'usage de tel ou tel paquetage générique pour l'interprétation des appels asynchrones sur un déploiement particulier pourra être spécifié, par exemple, à l'aide d'annotations sur les nœuds de déploiement.

Pour effectivement mettre en œuvre le paquetage `Generiques` sous la forme de procédures automatiques dans un AGL, il est bien entendu nécessaire de disposer d'un langage pouvant manipuler le métamodèle UML. Il y a des AGL dotés de cette capacité, mais les langages utilisés sont toujours propriétaires car il n'existe pas encore de langage normalisé pour cela. En conséquence, avant d'effectuer la programmation proprement dite des motifs récurrents de chaque méta-module, il est souhaitable d'en fournir d'abord un modèle UML, et ce, afin d'en préserver une trace et une spécification indépendantes des langages propriétaires. Les motifs récurrents correspondant essentiellement à des procédures applicables sur les modèles fonctionnels, les modèles du paquetage `Generiques` comprendront une large part de descriptions dynamiques sous la forme de diagrammes de séquence ou d'activité.

IV.C.3 Modélisation fonctionnelle

Il est clair que le paquetage `Fonctionnel` sera le premier abordé dans le processus de développement d'un système. En effet, la première activité des ingénieurs et physiciens dans un projet TDAQ consiste en réunions de réflexion libre et de *brainstorming* sur la définition des fonctionnalités du système. Cela correspond aux premières occurrences de l'étape de « description fonctionnelle » explicitée au IV.B.2 (fig. IV-7, page IV-15). Comment et dans quel ordre utiliser les diagrammes UML afin d'élaborer notre modèle fonctionnel et enrichir le paquetage associé ?

Le cheminement psychologique naturel dans une activité de développement TDAQ débute par la visualisation du substrat matériel du système au sens du IV.B.3

(p. IV-18) et du IV.B.4.2. Cette visualisation consiste essentiellement à décrire par des diagrammes de blocs (non-UML) les emplacements relatifs des différents nœuds de détection, les principaux flux de données et les topologies de communication possibles. Les figures IV-16 p. IV-28, IV-17 p. IV-32, IV-20 p. IV-44 et IV-21 p. IV-46 sont des exemples de tels diagrammes. Nous pourrions développer une graphie fondée sur UML[19] afin que la description de ce substrat soit conforme à UML mais nous ne sommes pas convaincus qu'elle aurait un pouvoir d'évocation suffisant pour être adopté par les ingénieurs. Cette première visualisation du substrat matériel du système permet d'orienter la réflexion sur les différents *objets* du système que ce substrat va supporter. Il s'agit de déterminer quels sont les objets principaux qui vont permettre de formater, transporter et traiter les flux de données physiques produits par les nœuds de détection.

IV.C.3.1 Cadre conceptuel du paquetage « Fonctionnel »

Ce qui nous intéresse dans cette phase est de définir en détail ce que nous attendons du système en termes de traitement et de transmission de données indépendamment des considérations de performances (CPU, réseau, etc.). Cela nous mène à poser une première règle de développement :

- Toute réflexion portant sur le paquetage `Fonctionnel` doit s'effectuer en supposant *l'accès à une puissance de traitement infinie*.

Nous ne devons, par exemple, pas introduire dans notre modèle fonctionnel des éléments de parallélisme de performance. En revanche, le parallélisme intrinsèque doit certainement être défini à ce niveau. De plus, la spécification des fonctionnalités du système ne doit pas s'embarrasser de détails concernant le mode de communication entre les objets. Ainsi, le seul mode de communication envisagé pour le paquetage `Fonctionnel` est l'*appel de méthode*, avec cependant, la possibilité de spécifier des appels asynchrones[20]. Notre deuxième règle de développement peut donc s'énoncer ainsi :

19. A base de nœuds, de paquetages « sous-systèmes », ou encore d'objets ou de composants.

20. Cette possibilité est nécessaire étant donné que nous avons du parrallélisme intrinsèque dès le niveau fonctionnel. A partir du moment où nous pouvons envisager des processus concurrents, nous devons autoriser l'usage de communications asynchrones au même titre que les communications synchrones.

- Toute réflexion portant sur le paquetage `Fonctionnel` doit s'effectuer en supposant que *tous les objets appartiennent à un même espace d'adresse abstrait*.

La transmission de paramètres lors des communications s'effectue aussi directement selon les modèles de programmation classiques, soit par valeur, impliquant alors une copie, soit par référence, par transmission d'un pointeur. Cependant, le type d'application qui nous intéresse ici nécessite d'introduire quelques précisions sur la sémantique du passage de paramètres lors des communications inter-objets.

IV.C.3.2 Modes de transmission des paramètres

Les applications TDAQ étant des systèmes d'acquisition de données, elles nécessitent que l'on puisse spécifier toutes caractéristiques permettant de nuancer la sémantique des flots de données. Etant donné que notre modèle de communication pour le paquetage fonctionnel est celui de l'appel de méthode, toute transmission de données sera effectuée naturellement à travers la transmission de paramètres dans ces appels. UML distingue 4 sortes de paramètres : `in`, `out`, `inout` et `return` ([B-11], p. 2-43), conformément au modèle le plus répandu dans les langages de programmation. Rappelons que les spécifications `in`, `out` et `inout` se rapportent à la la nécessité (ou la possibilité) pour la méthode appelée de modifier ou non le paramètre en question. En réalité, c'est une manière pour le développeur de la méthode appelée d'exprimer le type de paramètre *auquel il s'attend*. En termes imagés, nous pourrions dire que *l'appelant n'a pas son mot à dire* sur l'usage et la destination du paramètre.

Or, imaginons par exemple, qu'un objet `o1` veuille transmettre des données d'acquisition à un objet `o2` par appel de la méthode `o2.acq(d:Data)`, ces données étant regroupées dans l'objet `d`. Typiquement, une fois `d` transmis à `o2`, `o1` n'en a plus besoin et peut continuer ses autres traitements en *oubliant* `d`. Nous avons plusieurs manières d'interpréter cette communication dans un langage d'implémentation. Nous pouvons ainsi dire qu'une *copie* de `d` doit être transmise à `o2` (transmission par valeur) et que `o1` doit alors détruire sa propre copie de `d`. En C++, cela donne, par exemple, les instructions suivantes (en supposant que les variables d'objets sont toutes des pointeurs) :

```
o2->acq(*d); // transmission par valeur de l'objet Data
delete(d); // destruction de l'original
```

Notons que les types de paramètres définis par UML ne permettent pas de spécifier cela de manière non ambiguë. En effet, aucun d'eux ne permet de spécifier une transmission par valeur, le mot-clé `in` ne concernant que la question de la modification de `d` par `o2`[21]. Remarquons également que dans notre exemple, l'appel de méthode pourrait tout-à-fait être asynchrone, la destruction de l'objet `d` n'affectant en rien la copie reçue par `o2`. Cependant, le développeur d'un système TDAQ répugnera le plus souvent à adopter cette solution d'implémentation en raison de la quantité importante de données, des limitations mémoire, de la nécessité de ne pas gaspiller de la puissance processeur et autres contraintes de performances analogues. Il optera donc plutôt pour un scénario où seule une copie de la *référence* de `d` est transmise à `o2`, la copie de `o1` étant ensuite remise à zéro (ou écrasée par une référence ultérieure). En C++, cela donnerait :

```
o2->acq(d); //Transmission de l'objet Data par référence
d = null; // Effacement de la référence d
```

On peut dire en quelque sorte que dans ce scénario, la *responsabilité* de l'objet `d`, et notamment *celle de sa destruction*, est transférée de `o1` à `o2`. Dans le premier scénario d'implémentation envisagé, `o1` faisait usage de sa responsabilité de destruction de `d` en le détruisant effectivement et en transférant un nouvel objet à `o2`. Ainsi, selon que l'on adopte l'un ou l'autre scénario pour la transmission de l'objet `d`, `o1` doit procéder ou non à la destruction de sa copie de `d`. Par conséquent, la question de savoir si `d` doit être détruit après l'appel de la méthode `o2->acq()` relève de l'implémentation choisie pour ce mode de transmission. Plus précisément, la destruction de `d` après l'appel à `o2->acq()` *n'est pas* une spécification fonctionnelle, mais dépend de la manière dont on met en œuvre la notion de *transfert* d'objet. Il est donc indispensable de pouvoir spécifier le mode de transmission d'un paramètre au niveau fonctionnel sans avoir à fournir de précisions sur l'implémentation du transfert. Contrairement aux spécifications `in`, `out` et `inout`, cette spécification relève de l'objet appelant.

Précisons également qu'outre la notion de *transfert* d'objet, nous devons pouvoir

21. D'ailleurs, les implémentations C++ des générateurs de code transmettent le paramètre par référence et traduisent la spécification `in` *par le modifcateur* `const`.

spécifier les notions connexes de *partage* d'objet et de *duplication* d'objet. Le partage d'objet est spécifié lorsque `o1` tarnsmet l'objet `d` à `o2` afin d'en partager l'état et les données ; dans ce cas, `o1` ne doit pas effacer sa référence à l'objet `d` après l'appel de la méthode de `o2`. Ce mode d'appel correspond à la transmission la plus « naturelle », celle qui correspond au simple passage de paramètre par référence[22]. Quant à la duplication d'objet, elle correspond au cas où `o1` doit maintenir une copie de `d` indépendante de l'objet transféré à `o2` ; ce mode de transmission correspond bien entendu à la transmission par valeur.

Afin de pouvoir préciser sur les stimuli de nos modèles UML le mode de transmission des paramètres selon ces trois modalités, nous introduisons l'étiquette `transmission` susceptible de prendre les valeurs `share`, `move` et `copy`. Pour mettre en œuvre la modélisation comportementale des systèmes, la spécification UML définit, entre autres, les métaclasses `Instance`, `Stimulus`, `Action`, ainsi que les métaclasses dérivées de `Action` telles `CallAction`, représentant l'appel de méthode. Le métamodèle d'UML décrit l'articulation entre ces métaclasses permettant à l'utilisateur de spécifier un modèle comportemental (Cf. [B-11], pp. 2-88 à 2-105). Les figures 2-15 et 2-16 de la section *UML Semantics* du document de référence [B-11] de la spécification d'UML 1.3 nous permettent d'exhiber sur la figure IV-30 l'exemple d'une instance d'appel de méthode : le modèle d'un tel appel est représenté par une instance de stimulus lié à l'objet `CallAction` associé à l'opération invoquée, aux instances des objets appelé et appelant ainsi qu'aux instances des paramètres transmis lors de l'appel. Ce sont ces objets paramètres que nous nous proposons d'annoter avec l'étiquette `transmission`.

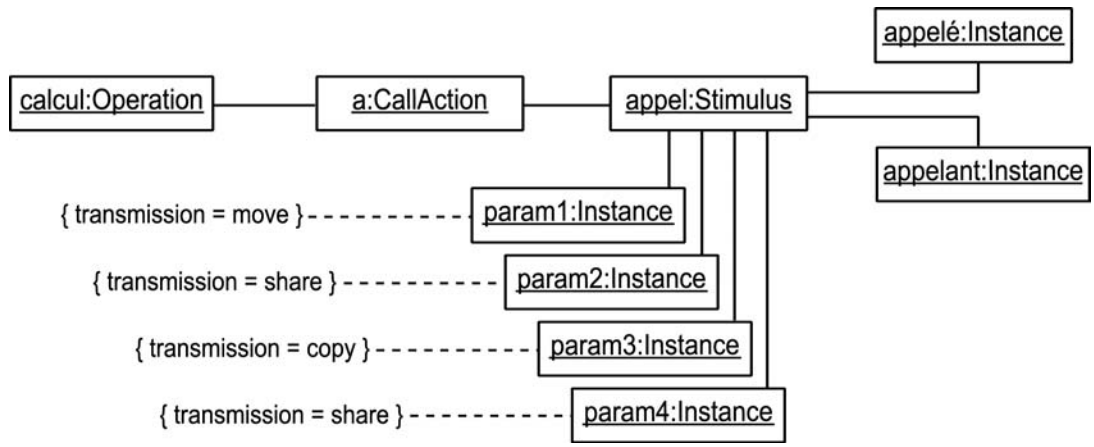


Fig. IV-30 : Instance d'un appel de méthode avec mode de transmission des paramètres annotés

22. Ce mode de passage de paramètre est d'ailleurs le seul mode utilisé pour les objets du langage Java, le passage par valeur étant réservé aux seules entités scalaires.

Bien entendu, il ne s'agit pas pour le développeur d'exhiber ainsi une représentation habituellement interne à l'AGL mais de procéder plutôt à ces annotations sur les diagrammes comportementaux d'UML, notamment les diagrammes de séquence. La transposition du diagramme en cette représentation interne étant du ressort de l'outil de développement ([B-11], pp. 3-101 à 3-103). La notation que nous proposons est celle de l'annotation de la flèche d'appel de méthode sur le diagramme de séquence par des expressions de la forme :

```
{<TypeTransmission>(<nomParamètre>),...}
```

où `<TypeTransmission>` prend l'une des valeurs (`share`, `move`, `copy`) et `<nomParamètre>` le nom du paramètre concerné dans l'appel, qui ne sera autre chose que le nom d'un objet ou d'un rôle à portée de l'objet appelant. Une illustration de ce mode d'annotation est donnée figure IV-31.

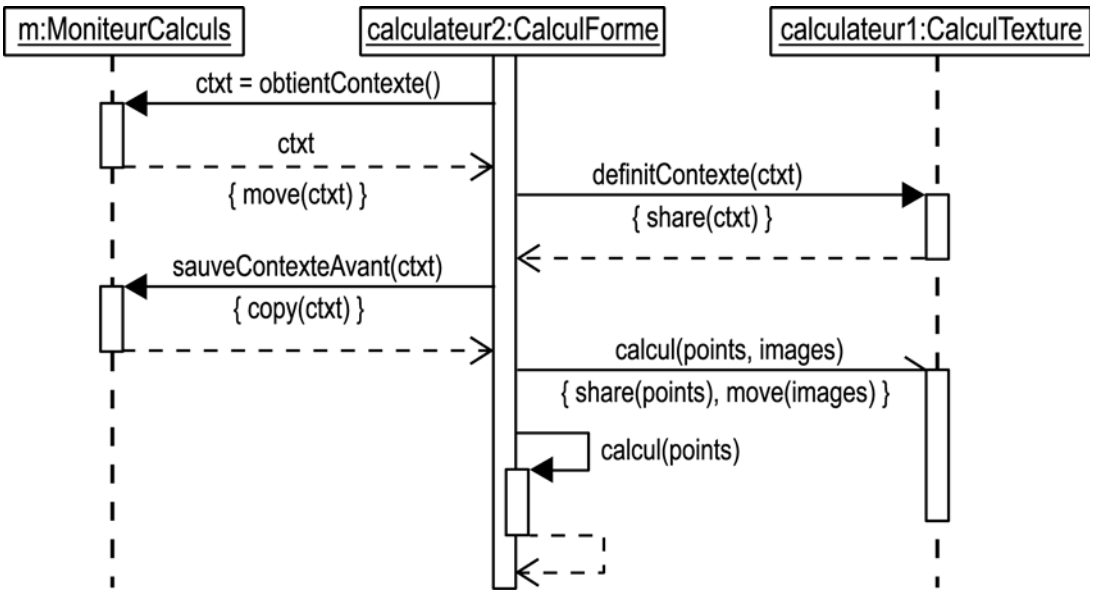


Fig. IV-31 : Séquence d'appel de méthode avec annotation de la transmission de paramètres.

Ajoutons également que le mode de transmission par défaut des paramètres sera `share` pour les objets et `copy` pour les scalaires (à l'instar du langage Java). Nous verrons au IV.C.4.1 que la spécification du mode de transmission est déterminante pour la mise en œuvre des motifs récurrents de communication entre objets distribués.

IV.C.3.3 Cas d'utilisation et développement TDAQ

Les méthodologies de développement les plus répandues de l'industrie ([C-33]) comportent une étape de spécification des fonctionnalités du système envisagé et

en soulignent fréquemment l'importance pour l'orientation future du travail, l'établissement des objectifs et la définition d'un cahier des charges (Cf. par exemple [C-33]). Celles qui utilisent la notation UML, comme le *Rational Unified Process* (RUP*) [C-32], préconisent souvent une utilisation étendue des diagrammes de cas d'utilisation (*use cases*) au cours de cette phase de spécification fonctionnelle, notamment pour établir une liste des interactions du système avec les acteurs extérieurs et des procédures que le système doit mettre en œuvre. Ces spécifications sont d'autant plus importantes qu'elle doivent aboutir à une intercompréhension approfondie entre développeurs et clients. La phase de définition des cas d'utilisation est donc en général une étape cruciale qu'il s'agit de ne pas bâcler au risque d'engendrer des malentendus entre prestataires et clients, voire des complications contractuelles. Cependant, nous ne pensons pas que cette façon de procéder soit adaptée au cas particulier des développements TDAQ.

Premièrement, le mode de fonctionnement des équipes impliquées dans ces développements n'est pas fondé sur une relation client-prestataire. Les développements instrumentaux liés aux expériences de physique sont généralement initiées par des « collaborations », elles-mêmes constituées d'équipes d'experts provenant de différents laboratoires dans le monde qui effectuent une démarche volontaire de participation en apportant chacun une part du financement ainsi que des moyens humains. Progressivement, la responsabilité de chaque sous-système est collégialement assignée par la collaboration à un laboratoire participant. Ce laboratoire devra donc concevoir et réaliser un sous-sytème conforme aux spécifications générales[23] élaborées dans le cadre des discussions au sein de la collaboration. Or, justement, l'équipe en charge du développement d'un sous-système est, dans la quasi-totalité des cas, celle-là même qui a été la plus impliquée dans l'élaboration de l'architecture et des spécifications générales du sous-système en question. Elle possède donc *a priori* une connaissance assez précise des fonctionnalités que le sous-système doit remplir et n'a donc pas besoin d'élaborer cette connaissance par une interaction avec un client. Elle est, en quelque sorte, son propre client. En d'autres termes, le développeur est lui-même maître de ses cas d'utilisation.

Deuxièmement, les fonctionnalités générales d'un système TDAQ ainsi que ses

23. Ces spécifications concernent essentiellement l'architecture générale du détecteur complet ainsi que les performances recherchées.

interactions avec les utilisateurs et les autres sous-systèmes sont, pour ainsi dire, toujours les mêmes et en nombre très limité (c'est, d'ailleurs, la raison pour laquelle le développement d'un canevas TDAQ présente un grand intérêt). Elles ont été pour l'essentiel décrites au II.A. Le travail de conception autour d'un développement TDAQ est donc avant tout l'élaboration d'une architecture détaillée, la mise au point de protocoles d'acquisition, de calcul et d'acquisition ainsi que la détermination de solutions d'implémentation matérielles et logicielles concrètes. Aussi, la phase de spécification fonctionnelle du système ne consiste-t-elle pas en la recherche de cas d'utilisation mais bien en l'élaboration d'une *architecture de fonctionnement général* s'appuyant sur une architecture matérielle essentiellement contrainte par des nécessités scientifiques (Cf. II.A et II.A.7).

Pour ces raisons, nous sommes convaincus que les diagrammes de cas d'utilisation, sur lesquels les méthodologies objet s'appuient en général fortement, ne présentent pas d'utilité réelle dans le cadre restreint d'un canevas TDAQ. Tout au plus pouvons-nous envisager de produire dans un but académique une description des fonctionnalités des systèmes TDAQ *en général* sous la forme de cas d'utilisation[24].

IV.C.3.4 Débuter par les diagrammes comportementaux

Ainsi que le suggère le terme de « fonctionnel » et à l'instar des méthodologies objet les plus répandues ([C-32]), nous préconisons d'engager la réflexion sur le système sur la base de spécifications comportementales. Cette préférence accordée à la description comportementale comme base de développement du système se justifie par le mode de fonctionnement psychologique de l'être humain : lorsque nous réfléchissons à un système, nous nous représentons avant tout des objets physiques en interaction les uns avec les autres. Les concepts UML qui traduisent le mieux cette disposition naturelle sont ceux d'« instance » et d'« interaction ». Les concepts structurants tels les classes, les paquetages ou les interfaces correspondent à des notions abstraites et, en tant que telles, peu favorables à la réflexion spontanée, au « premier jet ». Précisément, nous ne devons pas les produire par une réflexion *a priori*, mais plutôt les *abstraire* à partir d'une première description fondée sur des objets (instances) et des interactions.

24. En revanche, les cas d'utilisation peuvent se révéler précieux pour la spécification UML du canevas lui-même ainsi que la description du processus de modélisation.

Nous avons vu dans les sections précédentes que la description d'un système TDAQ peut se résumer à la spécification des flots de données, des topologies de réseau ainsi que des nœuds de traitement permettant de formater, rassembler, et traiter ces données. Une grande part de l'activité de développement va donc être dévolue à la spécification *précise* de *protocoles* permettant de mener à bien l'ensemble de ces activités avec les performances requises. La précision dans la description de l'enchaînement des actions est déterminante pour la détermination des performances du système. Cette nécessité de précision implique que parmi les diagrammes comportementaux d'UML, les diagrammes de séquence seront les plus adaptés pour la spécification comportementale des systèmes TDAQ. Bien entendu, les diagrammes de collaboration ne sont pas à exclure, notamment pour commencer à se fixer les idées, mais il est probable qu'elle doivent *in fine* aboutir à des équivalents séquentiels. En réalité, ainsi que le souligne la spécification de la notation UML ([B-11] p. 3-98), un diagramme de séquence implique l'existence d'une collaboration implicite ; il n'est donc pas interdit de commencer par l'explicitation de celle-ci.

Ajoutons enfin qu'un diagramme de séquence se prête plus facilement à la génération automatique de code, puisque les langages informatiques les plus répandus sont fondés sur l'exécution séquentielle d'instructions et d'appels de procédures. Nous définirons ainsi des diagrammes de séquence restreints dépourvus des ambiguïtés qui font obstacle à la génération automatique de code.

IV.C.3.5 Obtenir un modèle structurel

Si notre approche de la modélisation fonctionnelle préconise l'analyse comportementale comme base de développement, elle ne s'y restreint bien entendu pas. Il s'agit d'abstraire de notre description fonctionnelle comportementale une description en termes de classes qui formera la base de la génération du code effectif. L'élaboration de ce modèle appartient à la phase d'élaboration d'architecture décrite au IV.B (Cf. fig. IV-10 p. IV-18).

Ce modèle structurel est, comme dans tout développement orienté objet, appelé à être affiné à travers des factorisations, des regroupements et des découpages en vue d'assurer une modularité et une maintenabilité optimales du système TDAQ. Une fois ce modèle structurel fonctionnel obtenu, nous allons pouvoir passer à nos premiers déploiements afin de procéder à la production de modèles exécutables

par transformation du modèle fonctionnel.

IV.C.4 Déploiements et transformations du modèle fonctionnel

Nous avons évoqué au IV.C.2.2 le lien que nous établissons, dans un diagramme de déploiement, entre le nom des associations inter-nœuds et le paquetage générique spécifiant une mise en œuvre précise de la communication inter-objets. Nous avons également évoqué au IV.C.2.3 que ce type de lien pouvait être établi entre toute annotation des diagrammes de déploiement et des méta-modules génériques afin de produire des modèles exécutables à partir du modèle fontionnel. Dans cette section, nous allons détailler et discuter quelques exemples simples afin d'en déduire des règles d'annotation rigoureuses ainsi que les éventuelles extensions UML nécessaires. Les exemples de motifs récurrents évoqués sont tous inspirés de motifs effectivement appliqués au cas du système d'acquisition de l'expérience ANTARES.

IV.C.4.1 L'appel distant synchrone

IV.C.4.1.a Description de la problématique

Un exemple simplifié de spécification de module générique serait un motif d'appel de méthode synchrone distant de type CORBA ou Java RMI. Considérons un système simple de type « client-serveur » dont la modélisation fonctionnelle est représentée sur les diagrammes de séquence et de classe de la figure IV-32.

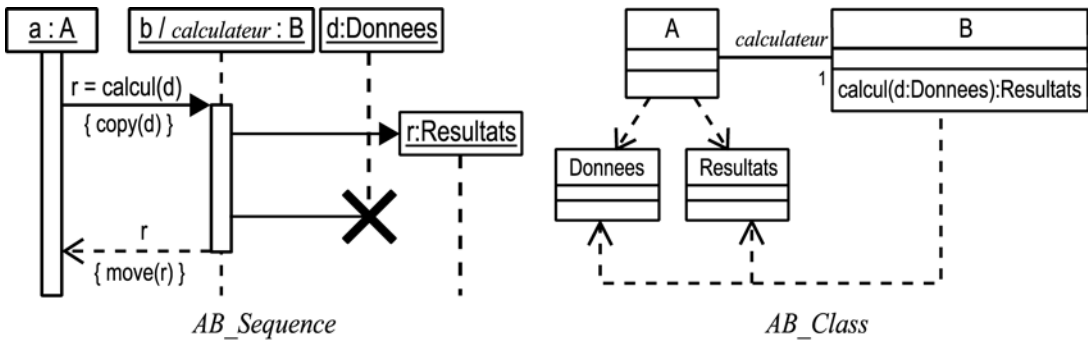


Fig. IV-32 : Diagrammes fonctionnels de séquence et de classe du projet AB

D'après la règle énoncée au IV.C.3 p. IV-67, le modèle fonctionnel se place dans le cadre d'un unique espace d'adresse. Cela siginifie en particulier que par défaut, l'appel synchrone de la méthode `calcul()` de l'objet `b/calculateur` équivaut, dans le modèle fonctionnel, à un classique appel de procédure. Considérons maintenant que ce modèle soit précisément appelé à être déployé sur deux

machine reliées par réseau. Le diagramme de déploiement d'une telle mise en œuvre est explicité sur la figure IV-33.

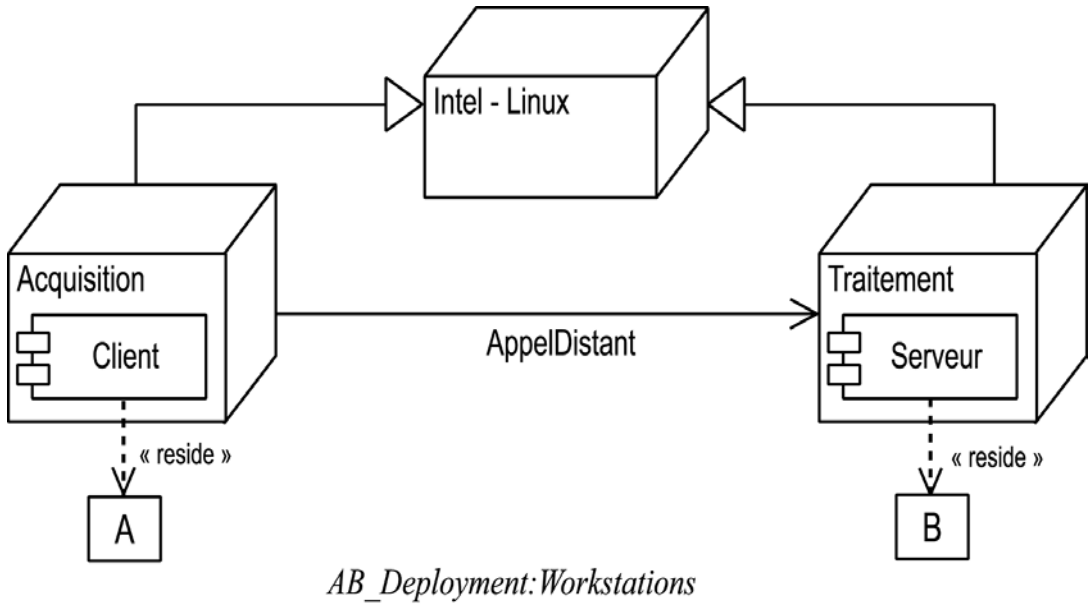


Fig. IV-33 : Déploiement du projet AB avec communication entre espaces d'adresse différents (nom du déploiement:Workstations)

Ce diagramme indique que l'application est formée de deux composants `Client` et `Serveur` implémentant respectivement les classes `A` et `B`. Les deux composants sont destinés à être déployés sur la même plateforme `Intel-Linux`, mais sur des machines (et donc des espaces d'adresse) différentes. Le modèle fonctionnel des classes `A` et `B` étant développé dans le cadre d'un espace d'adresse unique, nous ne pouvons le mettre en œuvre tel quel sur le déploiement proposé. Nous avons donc besoin d'une *procédure* pour *transformer* le modèle fonctionnel en un nouveau modèle permettant de fabriquer effectivement les composants `Client` et `Serveur`. Cette procédure de transformation doit correspondre à la mise en œuvre d'un *motif récurrent* traitant précisément de la problématique de distribution d'objets sur des espaces d'adresse distincts. Dans notre exemple, le nom de ce motif récurrent est `AppelDistant`, les deux diagrammes de déploiement y faisant référence au travers du nom de l'association entre deux nœuds `Intel-Linux`. Un méta-module générique homonyme devra décrire ce motif en détail. Notons également que l'association entre les nœuds `Acquisition` et `Traitement` sur le diagramme de la figure IV-33 est unidirectionnelle (navigabilité de `Acquisition` vers `Traitement`) car notre projet ne nécessite que des appels de méthode d'objets du composant `Serveur` par des objets du composant `Client` . Il est bien entendu que nous pourrions tout-à-fait spécifier un lien bi-

directionnel signifiant par là que les appels distants dans les deux sens doivent être mis en œuvre par le même méta-module `AppelDistant`. Notre notation autorise également l'usage de deux motifs différents, un pour chaque sens, auquel cas, nous devrions spécifier ces deux modules par deux associations de noms différents et de sens contraires.

IV.C.4.1.b Détail des procédures

Un exemple de procédure `AppelDistant`, que nous pourrions appeler plus précisément « `UneConnexionParMéthode` », serait l'établissement d'une connexion réseau associée à chaque méthode amenée à être appelée à distance dans notre application. Les figures IV-34 et IV-35 illustrent les résultats possibles d'une telle procédure. Celle-ci peut être résumée par les étapes suivantes :

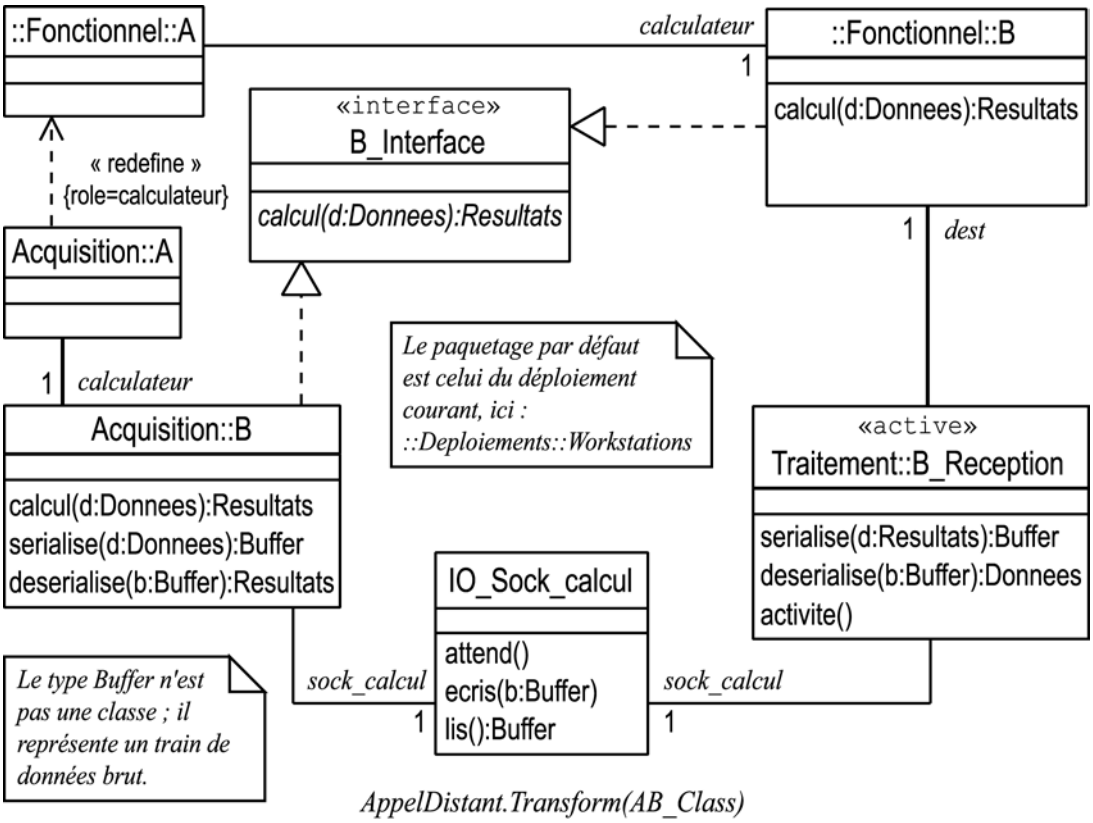


Fig. IV-34 : Diagramme de classe du projet AB transformé par le méta-module `AppelDistant`

1. Les objets dont une ou plusieurs opérations sont appelées à distance sont déterminés à partir des diagrammes de séquence et de déploiement ; nous appelerons ces objets les « objets cibles ». Dans notre exemple, seul l'objet `b`, jouant le rôle de calculateur pour l'objet `a`, est dans ce cas (fig. IV-32 et IV-33).
2. A partir de la classe de chacun de ces objets (les « classes cibles »), une interface n'incluant que les opérations en question est extraite. Dans notre exem-

ple, l'interface `B_Interface` est extraite de la classe cible `B` (fig. IV-32 et IV-34).

3. Chaque nœud du déploiement donne lieu à la création d'un sous-paquetage homonyme destiné à contenir les classes effectives nécessaires à l'instantiation des objets de l'espace d'adresse associé au nœud.

4. Pour chaque classe cible, une classe d'envoi de requêtes est créée, réalisant l'interface extraite de la classe cible, ainsi qu'une classe réceptrice[25]. Ces deux classes doivent implémenter des méthodes de sérialisation/désérialisation[26] d'objets, en vue de la transmission des paramètres à travers le réseau, entre espaces d'adresse différents. Dans notre exemple, les classes d'envoi et de réception associées à la classe cible `B` sont `Acquisition::B` et `Traitement::B_Reception` (fig. IV-34).

5. Dans chaque paquetage correspondant à un nœud de déploiement, toute classe appelant une classe cible est remplacée par une classe équivalente associée à la classe cible équivalente dans ce paquetage. Les caractéristiques (rôles, cardinalités,...) de ces associations équivalentes sont maintenues à l'identique. Dans notre exemple (fig. IV-34), la classe appelante `::Fonctionnel::A` associée à la classe `::Fonctionnel::B` donne lieu sur le nœud `Acquisition` à la classe `Acquisition::A` associée à la classe `Acquisition::B` (dépendance `«redefine»` liée au rôle calculateur de l'association entre `A` et `B`, au sens défini au IV.C.6.1)

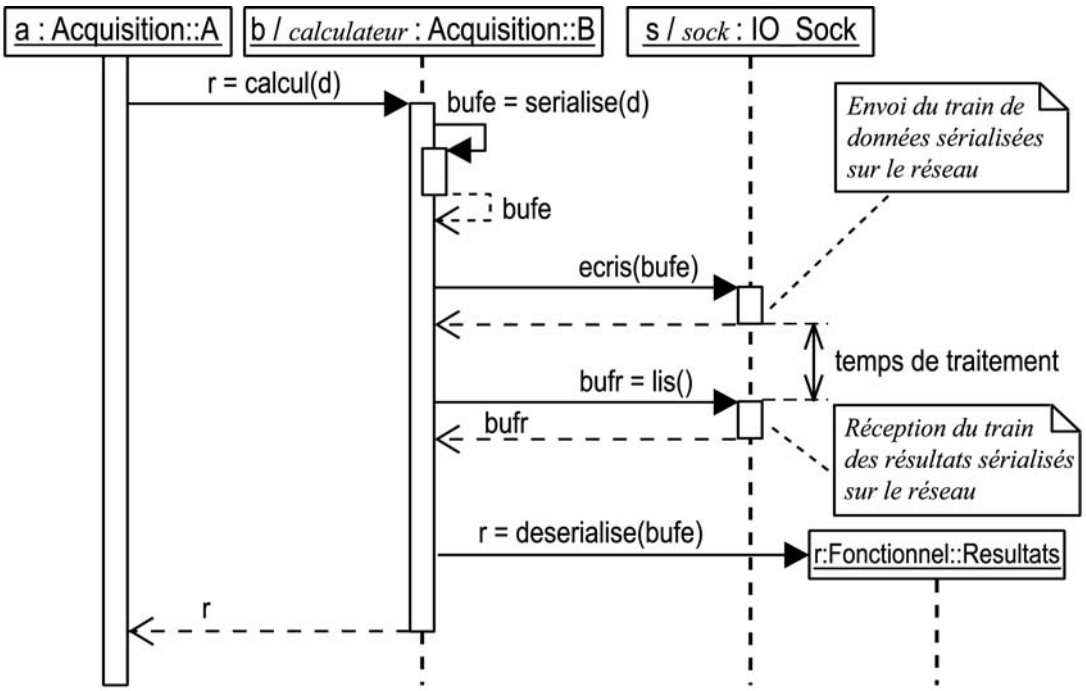
6. Chaque paire de classes d'envoi et de réception sont ensuite associées à une classe représentant un port d'entrées/sorties sur le réseau *pour chaque opération* déclarée dans l'interface. Cette classe implémente les fonctions de transmission de données à travers le réseau. Dans notre exemple, une seule opération est concernée :

   `B::calcul(Donnees):Resultats.`

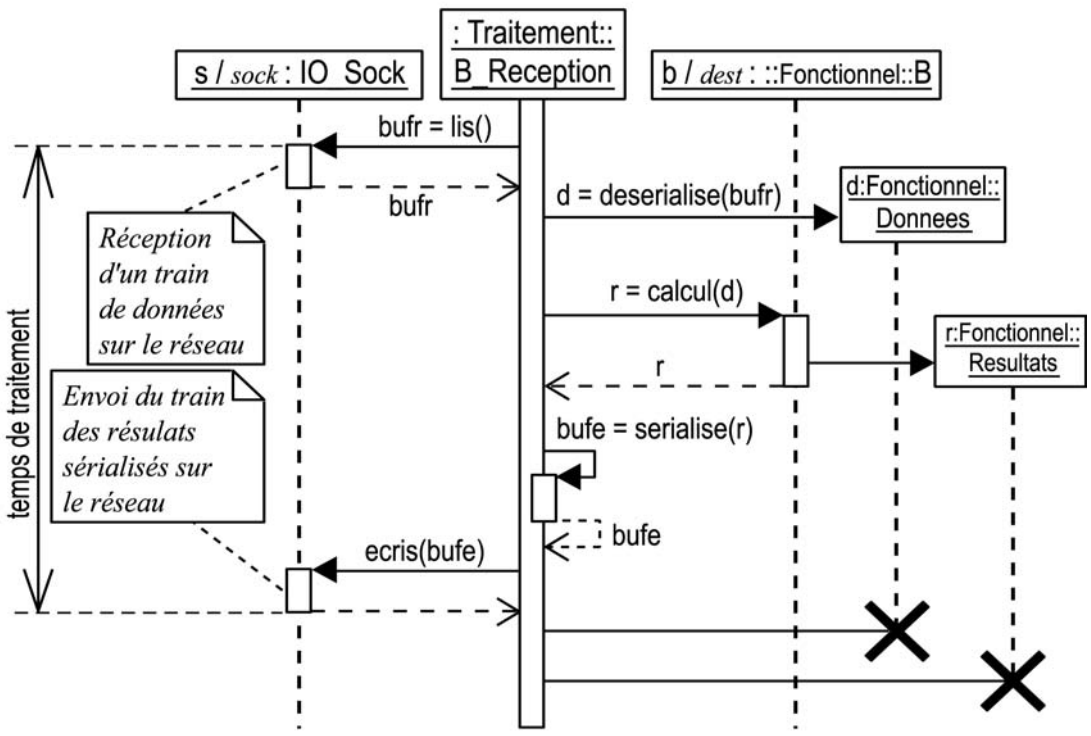
   une seule classe d'entrées/sorties (`IO_Sock_calcul`) est ainsi associée aux classes d'envoi et de réception (fig. IV-34). Son rôle ***`sock_calcul`*** est donc de transmettre les données sérialisées de tous les paramètres (y compris le paramètre de retour) à travers le réseau. Bien entendu les instances des classes

25. Dans le langage des ORB, ces classes sont respectivement le « Proxy ». et le « Skeleton ».*
26. Dans le langage des ORB, marshalling/unmarshalling..*

`Acquisition::B` et `Traitement::B_Reception` auront chacune une instance de `IO_Sock_Calcul` propre à laquelle elles seront liées (une connexion par opération).



AppelDistant.Transform(AB_Sequence) - 1 (séquence client)



AppelDistant.Transform(AB_Sequence) - 2 (séquence serveur)

Fig. IV-35 : Séquences du projet AB résultant du la transformation par le méta-module `AppelDistant`. Ces séquences sont directement assimilable à du code C++ ou Java.

7. Les diagrammes de séquence effectifs permettant une génération non ambiguë de code sont eux mêmes générés à partir des séquences fonctionnelles. Dans notre exemple, le code de la méthode `calcul(Donnees):Resultats` de la classe `Acquisition::B` peut être directement déduit du premier diagramme de séquence de la figure IV-35 ; le code de la méthode `activite()`, qui constitue la boucle principale de l'objet actif de classe `Traitement::B_Reception`, peut être directement déduit du deuxième diagramme de séquence de la figure IV-35.

Notons au sujet de la procédure 5 que les classes `A` et `B` respectivement définies dans les paquetages `Acquisition` et `Traitement` correspondent à une redéfinition des spécifications des classes `A` et `B` du paquetage `Fonctionnel` où l'on aura remplacé toute classe associée navigable, déployée sur un espace d'adresse différent, par une classe « *proxy* » implémentant la même interface. Dans notre exemple, l'association entre `::Fonctionnel::A et ::Fonctionnel::B` est remplacée dans le paquetage `Acquisition` par l'association entre `Acquisition::A` et `Acquisition::B` où ce dernier joue le même rôle pour `Acquisition::A` que joue `::Fonctionnel::B` pour `::Fonctionnel::A`. La relation de dépendance entre la classe d'origine (`::Fonctionnel::A`) et le résultat de sa transformation (`Acquisition::A`) est dénotée par le stéréotype `«redefine»` associé à une étiquette indiquant les rôles des associations redéfinies (fig. IV-34). La sémantique de ce stéréotype sera précisée au IV.C.6.

IV.C.4.1.c Importance des spécifications de transmission des paramètres

Le cas d'école que nous venons de détailler illustre également l'importance des spécifications de transmission des paramètres introduits au IV.C.3.2. Par exemple, la transmission de l'objet `r:Fonctionnel::Resultat` renvoyé par l'objet `b` a été annotée par la directive `move` (fig. IV-32) ; cela signifie que l'objet `b` n'en a plus besoin après sa transmission. Dans le cadre de notre déploiement sur deux espaces d'adresse différents, cette spécification se traduit par une destruction de l'exemplaire de `r` détenu par l'objet `b` après sa transmission à `a`, alors que dans un même espace elle n'aurait entraîné que l'écrasement du pointeur `r`, variable locale à la méthode `Acquisition::B::calcul(Donnees):Resultats`.

La transformation des séquences fonctionnelles décrite dans la 7^{me} étape de la

procédure de transformation de modèle constitue le cœur du motif récurrent implémenté par le méta-module `AppelDistant` : elle est la traduction concrète des protocoles mis en œuvre dans le méta-module pour fournir une interprétation particulière de l'appel distant synchrone.

IV.C.4.1.d Paquetages produits

Ainsi que nous l'avons annoncé au IV.C.2.2 (fig. IV-29), *in fine*, un paquetage par nœud de déploiement doit être produit, contenant chacun une partie des classes effectives ainsi que les diagrammes de séquence effectifs associés. Dans notre exemple, nous aurons bien entendu deux paquetages, l'un contenant les entités liées au composant `Client` et l'autre les entités du composant `Serveur`. Bien entendu, certaines entités telles la classe `IO_Sock_Calcul` ou l'interface `B_Interface` sont communes aux deux paquetages. Nous avons dans ce cas le choix de les dupliquer dans chaque paquetage ou, mieux, les regrouper dans le paquetage commun `::Deploiements::Workstations` associé au déploiement de la figure IV-33. La figure IV-36 représente l'organisation des paquetages du projet AB après la spécification du déploiement de la figure IV-33.

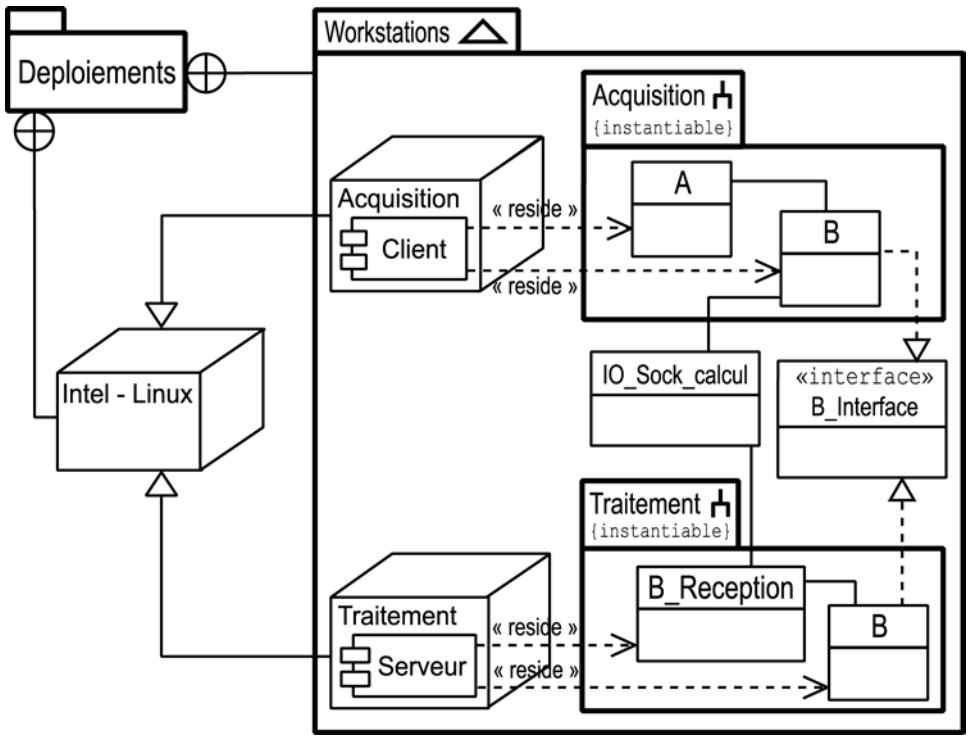


Fig. IV-36 : Organisation des paquetages effectifs de déploiement du projet AB

IV.C.4.1.e Communication avec des nœuds matériels

Rien ne nous empêche de mettre en œuvre un motif récurrent analogue à

`AppelDistant` pour des communications entre modules matériel et logiciel. Considérons par exemple toujours le projet AB, mais cette fois-ci déployé selon le diagramme de la figure IV-37. La communication inter-nœuds renvoie au méta-module `Interruption`. Celui-ci peut, à l'instar du méta-module `AppelDistant`, recourir à une classe `Traitement::B_Reception` dont la fonction serait de recevoir les requêtes d'exécution de la méthode `B::calcul(Donnees):Resultats` à travers une interruption matérielle et les paramètres (ici de type `Donnees`) à travers la mémoire partagée entre le FPGA* et le processeur. L'installation du vecteur d'interruption *ad hoc* se ferait naturellement dans le constructeur de la classe `Traitement::B_Reception`, lors de l'instantiation des objets de l'application.

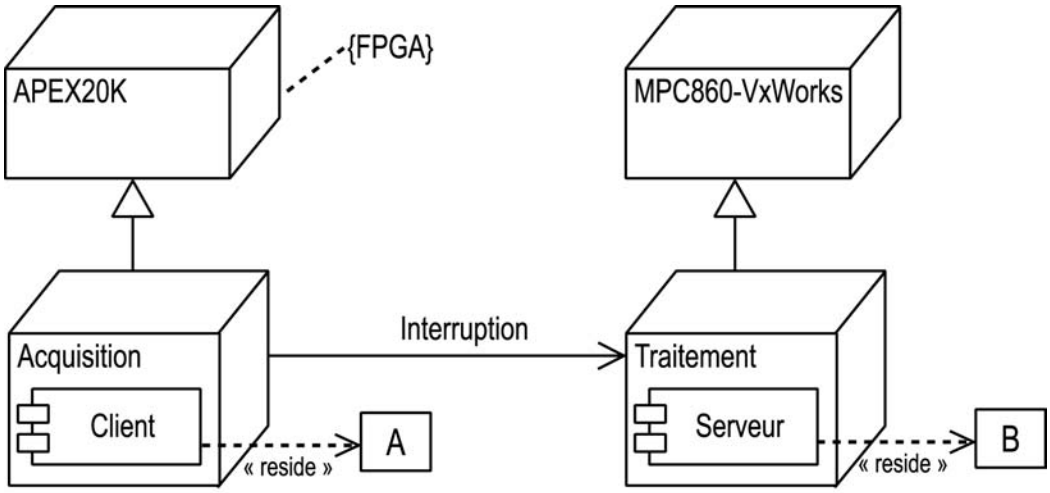


Fig. IV-37 : Exemple de déploiement comprenant une partie matérielle (nom du déploiement : OnBoard)

Côté composant `Client`, l'implémentation doit se faire en VHDL, mais il semble encore peu aisé de générer ce VHDL automatiquement à partir de diagrammes de séquence tels ceux de la figure IV-35. En effet, les spécificités des différents FPGA de marques différentes ainsi qu'un besoin presque systématique de concevoir des architectures électroniques pipelinées pour des questions de performances, interdisent pratiquement toute correspondance effective entre une séquence abstraite de type UML et un code VHDL efficace. Il demeure néanmoins la possibilité de générer les déclarations d'interface des composants VHDL[27] [C-34]. Il est également concevable de générer des références à des implémentations

27. En VHDL, cela s'appelle une « architecture ».

d'architectures déjà développées par ailleurs. Ainsi, rien n'empêche que notre canevas TDAQ comporte des « menus » de composants VHDL auxquels nous pourrions faire référence dans les spécifications de déploiement.

IV.C.4.2 L'appel asynchrone

IV.C.4.2.a Description de la problématique

Le fort parallélisme des applications TDAQ ainsi leur fort niveau d'interaction avec des modules matériels impliquent souvent l'usage de modules multitâches et de communications asynchrones. L'asynchronisme est une nécessité notamment lorsque l'application doit gérer de fortes fluctuations dans les flots de données (Cf. par exemple p. IV-21 et fig. IV-12). Or, la communication asynchrone ne fait pas partie des notions implémentées dans les langages de programmation les plus répandus, notamment C++ et Java. Par conséquent, si notre spécification fonctionnelle comprend de tels appels, il est nécessaire que tout déploiement associé renvoie, au moins pour chaque type de nœud, à un méta-module d'interprétation de l'appel de méthode asynchrone.

Les motifs récurrents qui traitent de l'implémentation des appels asynchrones se réfèrent toujours, sous des noms différents, à trois notions incontournables : la queue de requêtes, l'objet méthode et l'objet futur (Cf., par exemple, [C-11] [C-36]).

La première désigne une queue, liée à l'objet appelé, destinée à recevoir les requêtes d'exécution de méthode. En effet, l'appel asynchrone nécessite que l'objet appelant puisse déposer une requête d'exécution de méthode puis continuer sur son propre fil d'exécution sans avoir à attendre le retour de la méthode appelée. La manière d'implémenter cette queue d'exécution et notamment d'ordonnancer les requêtes varie à l'infini selon les besoin de l'application.

La deuxième notion est celle d'« objet méthode » (*method object*) appelée parfois « foncteur ». Elle désigne un objet qui représente une requête d'exécution de fonction, renfermant également en général les paramètres de l'exécution. Les foncteurs du langage C++ sont également une tentative de représentation de cette notion(Cf. [C-35] pp. 795-801). C'est l'objet méthode qui est destiné à être déposé dans une queue de requêtes de l'objet appelé.

La troisième notion est celle d'« objet futur ». C'est l'objet destiné à recevoir l'éventuelle valeur de retour de la méthode appelée. C'est une sorte de « boîte à messages » dans laquelle l'objet appelé doit déposer le résultat d'exécution, à charge pour l'objet appelant d'y puiser ce résultat selon ses besoins. Généralement, l'objet futur est associé à un sémaphore afin de le protéger de toute lecture antérieure à la complétion de la méthode appelée. Remarquons aussi qu'un tel objet futur peut être nécessaire même en l'absence de valeur de retour, lorsque la complétion de l'exécution de la méthode appelée doit être signifiée à l'objet appelant (dans ce cas, seule la fonctionnalité de sémaphore est utilisée).

De nombreuses plateformes de développement utilisent ces trois notions sous des formes diverses afin de mettre en œuvre des mécanismes d'appels asynchrones. La plateforme ACCORD, par exemple, développée par le LIST* au CEA*, comprend un exécutif dont la tâche est de recueillir les requêtes asynchrones et les objets futurs transmis par les objets appelants, de traiter les requêtes selon une politique d'ordonnancement paramétrée par des contraintes temporelles et enfin, de renseigner les objets futurs à l'issue de l'exécution des requêtes. Dans ce modèle de programmation (en C++), il revient au développeur de signifier l'appel asynchrone en procédant à l'allocation des requêtes d'exécution et objets futurs concernés [C-14]. Un autre exemple de modèle, plus simple mais moins élaboré (notamment en ce qui concerne les spécifications temps réel) est proposé par le canevas ACE sous la forme d'objets prédéfinis extensibles (*ACE_Task*, *ACE_Future*,...) pouvant être mis en œuvre selon le motif de conception « *Active Object Pattern* » explicité dans la documentation associée ([B-22], pp. 58 à 65).

IV.C.4.2.b Contraintes conceptuelles et notation

Selon les besoins des applications, les développeurs doivent être libres d'utiliser pour l'implémentation de leurs appels asynchrones les motifs les plus adaptés. Aussi, dans le cadre de notre canevas TDAQ, chaque implémentation effectivement développée représentera-t-elle un méta-module d'appel asynchrone particulier qui sera réutilisable par les autres applications, mais le choix ne se restreindra pas à ceux-ci puisque la structure du canevas autorisera toujours l'adjonction d'un nouveau méta-module implémentant une interprétation différente. La question qui se pose ici est : « comment indiquer le méta-module des appels asynchrones sur un diagramme de déploiement ? ». L'appel asynchrone est bien entendu une

notion concernant la communication entre objets et il serait donc naturel qu'à l'instar des méta-modules concernant les appels distants, nous représentions la référence à un méta-module d'appel asynchrone par un lien nommé entre deux entités. Afin de le distinguer d'un lien de spécification d'appel distant, nous ferons usage du stéréotype «`async`». Par souci de symétrie, nous introduisons également le stéréotype suivant pour le lien de spécification d'un appel distant : «`remote`», sachant que nous retiendrons ce dernier comme valeur par défaut d'un lien non annoté.

Une bonne factorisation de ces deux concepts de modélisation (appels distant et asynchrone) nécessite que les spécifications concernant les appels asynchrones n'interfèrent pas avec celles concernant les appels entre espaces d'adresse différents. Aussi devons-nous *restreindre toute spécification d'appel asynchrone à un seul espace d'adresse*. Toute spécification d'appels asynchrones entre deux espaces d'adresse différents devra donc résulter de la *composition* d'une spécification d'appel distant et d'une spécification d'appel asynchrone. Nous discuterons au IV.C.5 de la question des compositions entre spécifications de déploiement et nous en verrons des exemples. Pour l'heure, nous posons comme principe de modélisation que tout lien de spécification d'appels asynchrones dans un diagramme de déploiement doit se restreindre à des entités appartenant au même espace d'adresse. Jusqu'ici, nous avons implicitement supposé qu'un nœud UML (*Node*) représentait l'espace d'adresse. Il est cependant possible que dans certains cas, nous ayons besoin de recourir à une granularité plus fine en travaillant directement au niveau des composants voire des objets ; nous traiterons de cette question au IV.C.6. Pour l'instant, dans ce qui va suivre, un espace d'adresse sera représenté par l'entité « Nœud ».

IV.C.4.2.c Détail des procédures

Le motif d'interprétation des appels asynchrones que nous allons présenter ci-après à titre d'exemple correspond à celui que nous avons élaboré et mis en œuvre dans le projet ANTARES. Ce motif présente quelques caractéristiques particulières répondant à des exigences de performances. Nous avons choisi par exemple d'affecter une queue indépendante à chaque méthode appelée de manière asynchrone afin d'une part, de supprimer un niveau d'indirection (la requête ne doit pas être sondée pour déterminer quelle méthode doit être exécutée), et d'autre

part de ne pas créer de blocage d'un type de requête par un autre (problème analogue à l'inversion de priorité causée par le traitement de tâches de priorités différentes à travers une queue unique, cf. [B-31]). Nous avons également décidé de réaliser l'objet méthode et l'objet futur à l'aide d'une unique classe, d'une part, afin de réduire les opérations de traitement de ces objets et d'autre part, parce que ces objet sont conceptuellement très liés puisqu'ils marquent respectivement le début et la fin de l'exécution d'une méthode. Pour illustrer le fonctionnement de ce motif d'appel asynchrone dans le cadre d'un déploiement particuler, reprenons l'exemple du projet AB légèrement modifié, appelé ABbis (fig. IV-38).

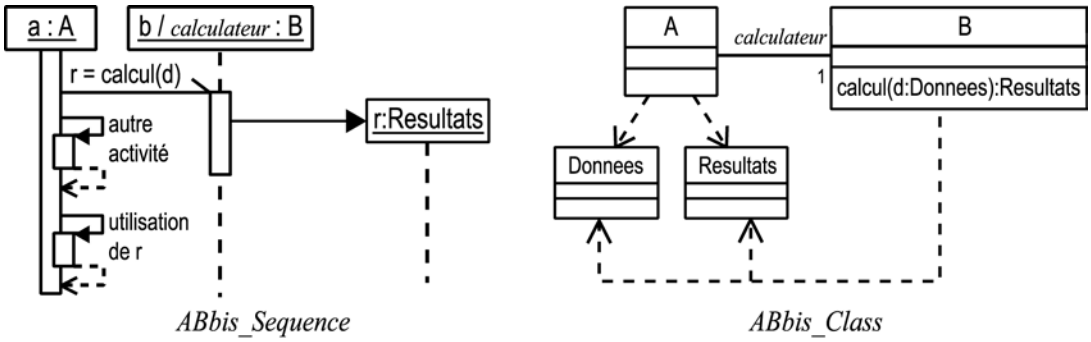


Fig. IV-38 : Diagrammes fonctionnels de séquence et de classe du projet ABbis

Le diagramme de séquence `ABbis_Sequence` présente un appel asynchrone de la méthode `b.calcul(d)` suivi d'une autre activité que l'objet `a` est donc susceptible de pousuivre sans attendre l'exécution complète de `b.calcul(d)`. Après cette activité, la variable r, contenant la valeur de retour de la méthode `b.calcul(d)`, est utilisée. L'analyse de la séquence montre donc que ce n'est qu'après l'achèvement de cette activité que le fil d'exécution de l'objet `a` doit subordonner la poursuite de son exécution à l'achèvement de la requête d'exécution de `b.calcul(d)`.

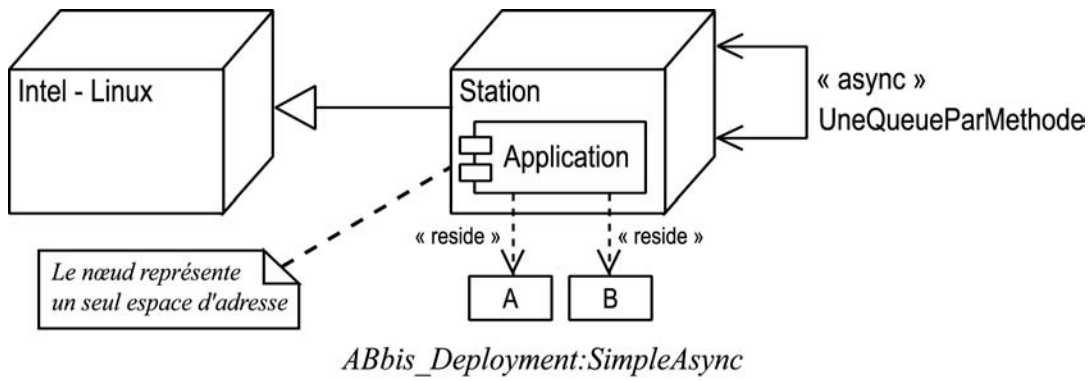


Fig. IV-39 : Déploiement du projet ABbis sur un seul espace d'adresse avec spécification de méta-module d'appels asynchrones

Supposons que nous appelions le motif récurrent mis en œuvre « `UneQueueParMéthode` ». La figure IV-39 montre le diagramme du déploiement `SimpleAsync` renvoyant à ce motif pour tout appel asynchrone sur le nœud `Worksation` de type `Intel-Linux`. La procédure de transformation du modèle fonctionnel par ce motif peut être décomposée comme suit :

1. Les « objets cibles » dont une ou plusieurs opérations sont l'objet d'appels asynchrones sont déterminés à partir des diagrammes de séquence et de déploiement. Pour chaque classe cible (classe d'objet cible) `C`, l'ensemble (noté A_C) des opérations de la classe appelées au moins une fois de manière asynchrone dans les diagrammes de séquence est déterminé. Dans notre exemple (fig. IV-38 et IV-39), seul l'objet `b`, jouant le rôle `calculateur` pour l'objet `a`, est dans ce cas et une seule opération de la classe B est concernée : `calcul(d:Donnees):Resultats`.

2. Pour toute opération $\omega \in A_C$, une classe `OM(`ω`)` est créée, dérivant d'une classe abstraite générale `ObjetMethode` définie par le méta-module d'appel asynchrone (fig. IV-40). `OM(`ω`)` hérite donc du sémaphore associé à `ObjetMethode` ainsi que la méthode `attendFin()` dont l'appel est bloquant jusqu'à ce que le sémaphore soit libéré. `OM(`ω`)` implémente la méthode `fin()` dont la fonction est d'attendre la libération du sémaphore pour renvoyer ensuite la valeur de retour de la méthode ω. Dans notre exemple : `OM(`ω`) = OM(calcul(d:Donnees):Resultats)= B_Calcul_OM`.

3. Pour chaque paramètre de la méthode ω, un attribut de même type est créé dans la classe `OM(`ω`)` afin d'y recevoir les instances des paramètres lors des appels asynchrones. De la même façon, si la méthode ω possède un paramètre de retour, un attribut de même type est associé à `OM(`ω`)`. Dans notre exemple (fig. IV-40), les classes `Donnees` et `Resultats` sont ainsi associées à `B_Calcul_OM` avec les rôles respectifs `d` et `valRetour`.

4. Pour toute opération $\omega \in A_C$, une classe d'objet actif `OA(`ω`)` associée à une queue d'objets méthodes est créée[28]. L'activité de `OA(`ω`)` se résume à une boucle de traitement des requêtes d'exécution de la méthode ω arrivant par la

28. L'implémentation de l'objet actif peut également faire l'objet d'un méta-module notamment si l'on veut en spécifier finement le comportement temps réel, mais nous ne développerons pas dans ce travail cet aspect, n'ayant besoin que de la notion minimale de « tâche », largement implémentée dans tous les systèmes d'exploitation actuels.

queue d'objets méthodes. Celle-ci correspond à un classe particulière fournie par le méta-module d'appel asynchrone. Dans notre exemple (fig. IV-40) : `OA(ω)= B_Calcul_Tache`.

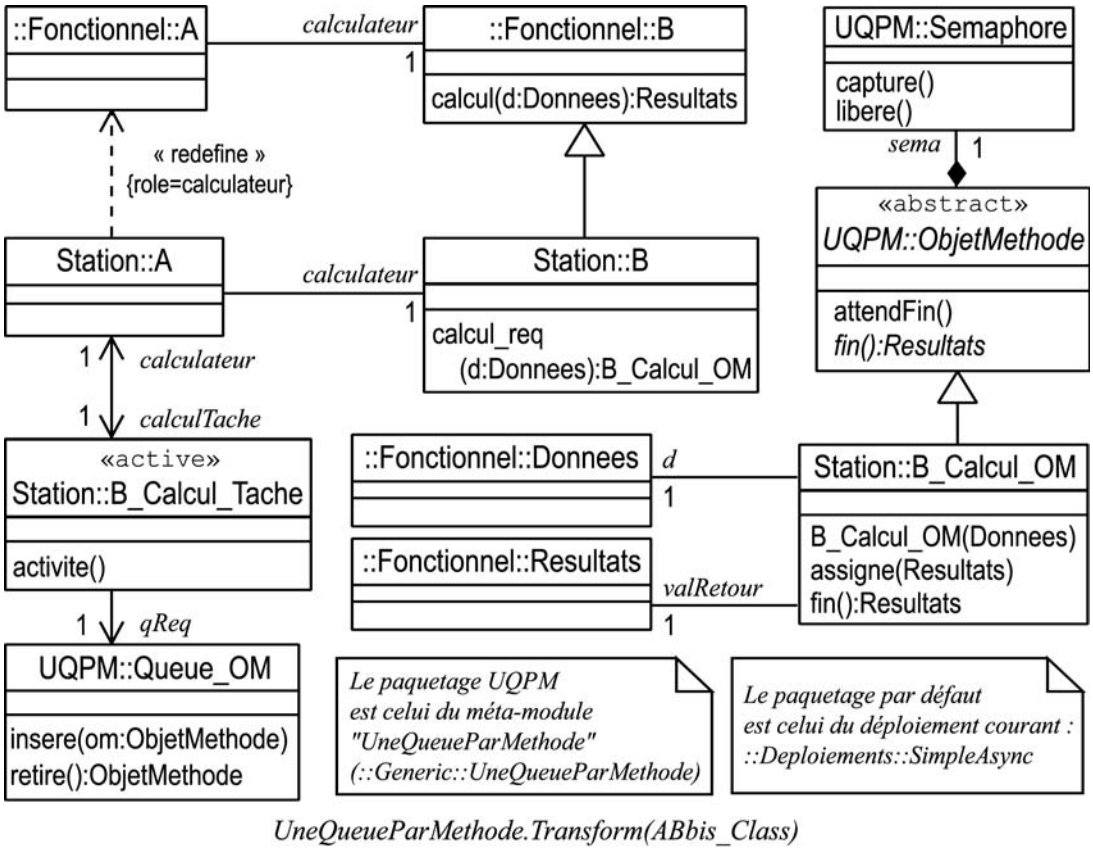
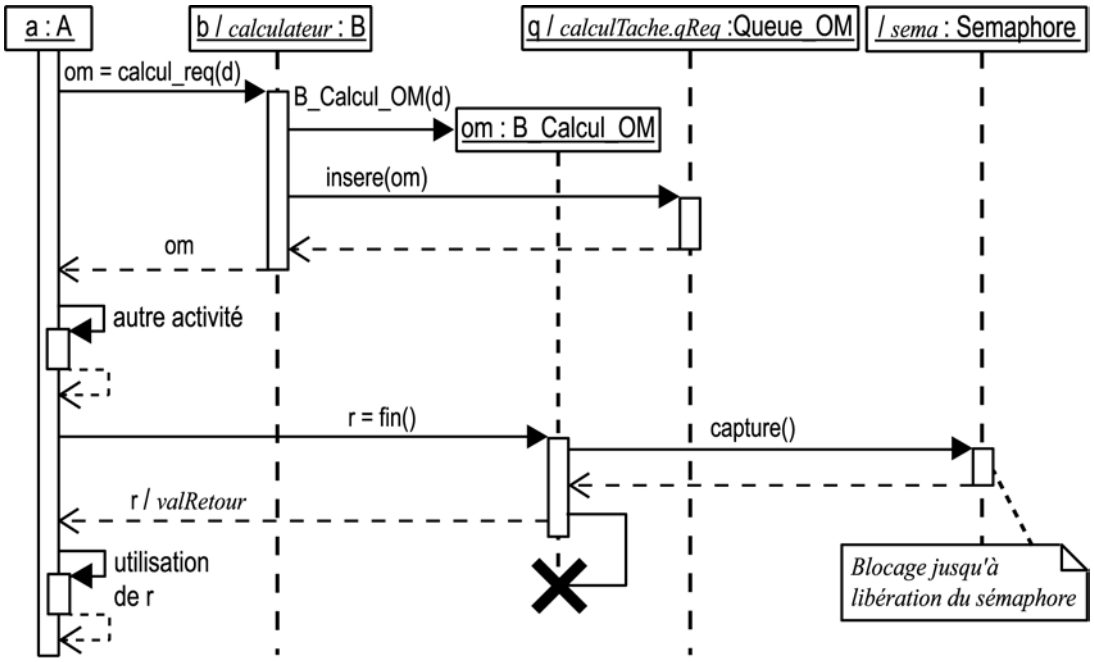
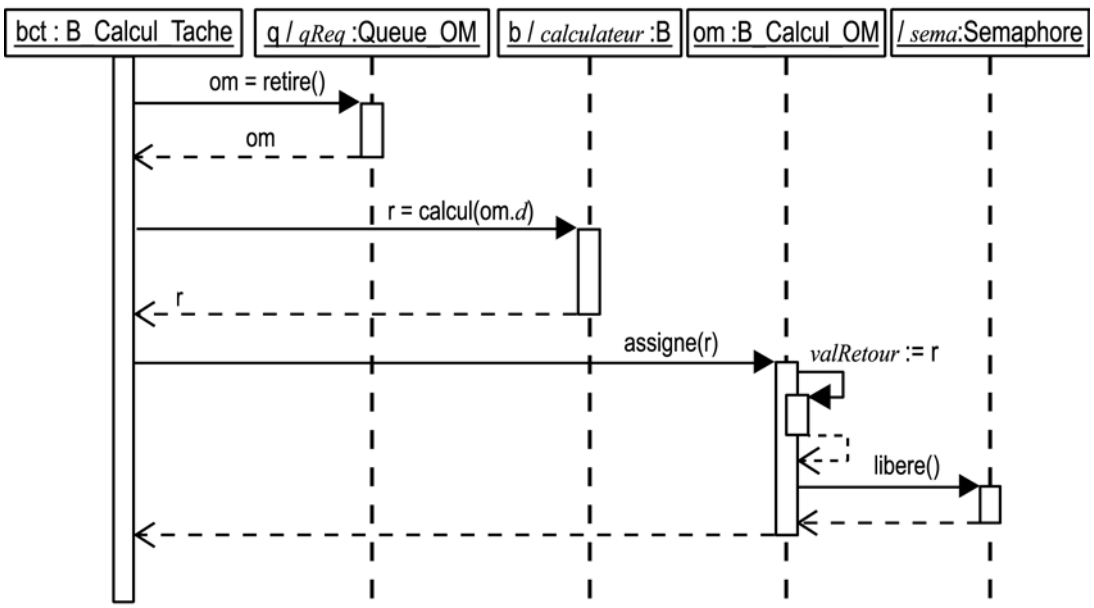


Fig. IV-40 : Diagramme de classe du projet ABbis résultant de la transformation du diagramme fonctionnel par le méta-module `UneQueueParMéthode`.

5. Pour toute opération $\omega \in A_C$, une nouvelle classe `C'` est dérivée de `C` et une opération `REQ(ω)` est ajoutée à la classe `C'`, dont la fonction est de créer la requête d'appel à ω, de l'insérer dans la queue de requêtes associée à `OA(ω)`. Le nom de `REQ(ω)`est formé du nom de ω concaténé avec le suffixe `_req` pour suggérer une « requête » d'opération. La signature des paramètres d'*appel* de `REQ(ω)` est identique à celle de ω ; en revanche `REQ(ω)` renvoie toujours un objet méthode de type `OM(ω)`. Dans notre exemple :
`REQ(ω)= calcul_req(d:Donnees):B_Calcul_OM`



UneQueueParMethode.Transform(ABbis_Sequence) - 1 (séquence requête)



UneQueueParMethode.Transform(ABbis_Sequence) - 2 (séquence exécution)

Fig. IV-41 : Séquences du projet ABbis résultant du la transformation par le méta-module `UneQueueParMéthode`. Ces séquences sont directement assimilable à du code C++ ou Java.

6. Dans le paquetage correspondant au nœud de déploiement, toute classe appelant une classe cible est remplacée par une classe équivalente associée à la classe cible équivalente `C'` dans ce paquetage. Les caractéristiques (rôles, cardinalités,...) de ces associations équivalentes sont maintenues à l'identique. Dans notre exemple (fig. IV-40), la classe appelante `::Fonctionnel::A`

associée à la classe `::Fonctionnel::B` donne lieu sur le nœud `Station` à la classe `Station::A` associée à la classe `Station::B` (dépendance `«redefine»` liée au rôle calculateur de l'association entre `A` et `B`, au sens défini au IV.C.6.1).

7. Les diagrammes de séquence effectifs directement interprétables pour la génération de code sont produits par le méta-module selon le protocole souhaité. Pour ce qui est de notre exemple, la figure IV-41 représente la séquence de création et d'envoi d'une requête d'exécution ainsi que la séquence de réception et d'exécution de la requête. Cette dernière constitue le cœur de la boucle d'activité de l'objet actif de classe `B_Calcul_Tache` associé à la méthode `B::calcul(Donnees):Resultats`.

IV.C.4.2.d Remarques sur la gestion de la valeur de retour

Il est nécessaire que l'outil de transformation autorise une analyse suffisamment fine des séquences pour pouvoir déterminer le nom de la variable recevant la valeur de retour de l'appel asynchrone (dans notre exemple " `r` ") ainsi que le point dans la séquence où cette valeur est utilisée pour la première fois, afin d'insérer la lecture bloquante de l'objet futur (dans notre exemple " `om` ") le plus tard possible après le dépôt de la requête d'appel (dans notre exemple, après " `autre activité` "). Une autre possibilité (fig. IV-42) est d'imposer au développeur d'expliciter sur le diagramme fonctionnel le point d'utilisation de l'objet futur par une flèche de retour de l'appel asynchrone (en pointillés). Cette dernière solution est d'autant plus satisfaisante qu'elle autorise la spécification de l'attente de l'objet futur même si la méthode ne retourne aucune valeur ; dans ce cas, la flèche de retour équivaut à un simple point de synchronisation avec la complétion de l'appel asynchrone.

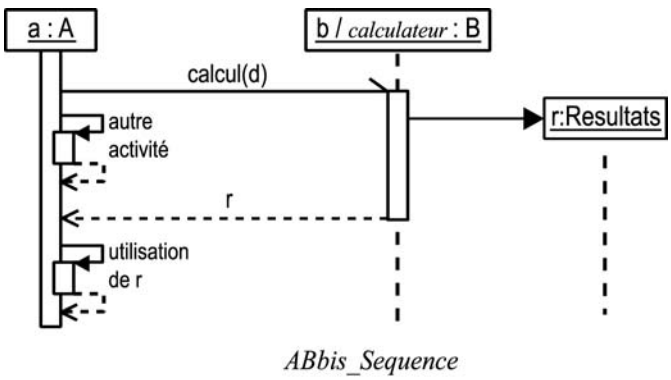


Fig. IV-42 : Exemple de séquence fonctionnelle avec point d'utilisation de la valeur de retour d'un appel asynchrone spécifié par une flèche de retour (en pointillés)

Enfin, nous devons traiter les cas d'« asynchronisme pur », c'est-à-dire où l'objet appelant n'a pas besoin d'accéder à l'objet futur. Dans ce cas, le schéma présenté pose en effet un problème puisque nous voyons figure IV-41 que la désallocation de l'objet futur (qui est également l'objet méthode) n'intervient qu'après l'appel de la sa méthode `OM::fin()` par l'objet appelant. Aussi, dans le cas où cet appel n'a pas lieu, l'objet méthode n'est-il jamais désalloué. Outre le problème de fuite de mémoire que cela pose pour les langages dépourvus de gestion de mémoire automatisée, l'absence de traitement spécifique de l'asynchronisme pur nuit également à l'optimisation des performances. En effet, un sémaphore est toujours alloué et au moins partiellement utilisé, dans ce cas en pure perte. Pour éviter cela, nous devons, là encore, spécifier au niveau fonctionnel si l'appel est purement asynchrone ou bien nécessite la production d'un objet futur. Pour cela, il suffit simplement de ne pas spécifier de valeur de retour pour cet appel (pas de flèche de retour). Le méta-module devra dans ce cas recourir à une version plus simple de la procédure de transformation qui ne crée pas de sémaphore et détruit l'objet méthode tout de suite après l'exécution de la méthode. Dans notre exemple (fig. IV-41) l'objet `bct` doit procéder à cette destruction au lieu de l'appel de la méthode `OM::assigne()`.

IV.C.4.3 Le déploiement en ferme

IV.C.4.3.a Description de la problématique

Nous avons vu au IV.B.3, p. IV-17, ainsi qu'au IV.B.7, p. IV-53, que certaines cardinalités plurielles dans un diagramme de classe effectif correspondaient à l'introduction de parallélisme de performance (tel que défini au IV.A.1). Nous avons également vu que ce type de parallélisme prenait, dans les systèmes TDAQ, le plus souvent la forme de fermes de traitement et impliquait donc la mise en œuvre d'une politique de gestion de cette ferme (Cf. IV.B.6, p. IV-49).

Par ailleurs nous avons explicitement exclu toute forme de spécification liée à des considérations de performances au niveau de l'élaboration fonctionnelle de nos applications TDAQ (Cf. IV.C.2.1et IV.C.3.1). Par conséquent, seules des spécifications de déploiement sont à même d'introduire dans les diagrammes effectifs les modifications nécessaires à la mise en œuvre des fermes de traitement. Cela nous amène à introduire un troisième type de méta-module, lié à l'implémentation du parallélisme de performance.

IV.C.4.3.b Analyse conceptuelle et notation

La mise en œuvre du parallélisme de performance équivaut à la « démultiplication » d'un objet du modèle fonctionnel (ou d'un groupe d'objets) sur plusieurs unités de traitement. Il s'agit donc de pouvoir spécifier, au niveau d'un déploiement particulier, une telle démultiplication. De plus, cette démultiplication n'est pas relative aux rapports qu'entretien l'objet avec telle ou telle composante de l'application : la démultiplication d'une entité affecte ses rapports avec toutes les autres entités de l'application. L'annotation permettant de spécifier le parallélisme de performance dans un déploiement doit donc s'appliquer à un objet ou un regroupement d'objets et non plus à un lien ou à une association entre différentes entités. En revanche, cette annotation affectera l'ensemble des associations qu'entretien l'entité avec les autres entités de l'application. L'annotation doit également permettre de renvoyer à un méta-module particulier de mise en œuvre d'une « ferme » de traitement. Ainsi, arrive-t-on à la conclusion naturelle d'annoter l'entité démultipliée par un stéréotype associé à une étiquette* renvoyant au nom du méta-module générique d'implémentation de la ferme. Le stéréotype que nous utilisons est «`farm`» et le nom de l'étiquette est `policy`, c'est-à-dire que tout composant de type «`farm`» doit être annoté par une valeur d'étiquette de la forme : `{policy = <`*`nom-metamodule`*`>}` (Cf. fig. IV-44).

Précisons également qu'à l'instar de l'appel de méthode distant, il s'agit de rendre cette démultiplication transparente par rapport au reste de l'application qui doit toujours voir l'entitée démultiplié comme une seule et unique entité. Enfin, pour les mêmes raisons de bonne factorisation évoquées p. IV-84, un motif récurrent de pur déploiement en ferme devra s'appliquer à l'intérieur d'un même espace d'adresse (comme pour du parallélisme en *multithreading*), le cas de la démultiplication sur plusieurs espaces d'adresse devant être obtenu par composition avec le motif d'appel distant (Cf. IV.C.5) ; aussi n'appliquerons-nous dans un premier temps l'annotation «`farm`» qu'à des composants *à l'intérieur* d'un nœud. Nous étendrons l'annotation au nœud dans le cadre de la composition des motifs (Cf. IV.C.6). L'appel asynchrone d'une méthode d'un objet démultiplié sera également obtenu par composition, les procédures présentées ci-dessous se restreignant à l'appel synchrone.

IV.C.4.3.c Détail des procédures

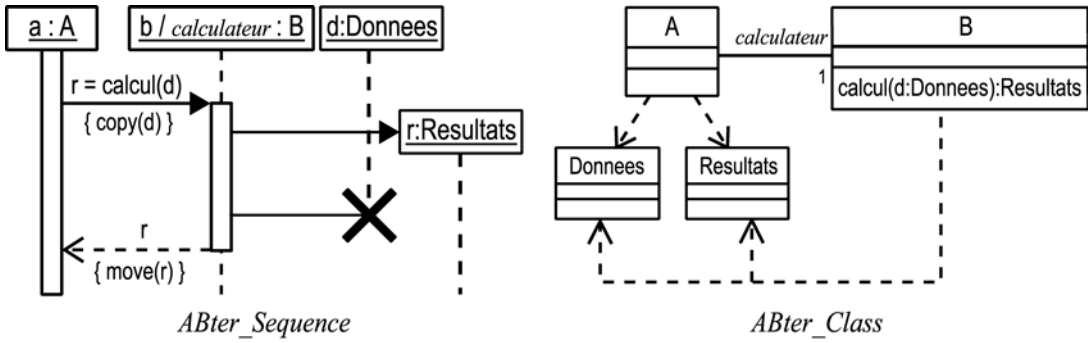


Fig. IV-43 : Diagrammes fonctionnels de séquence et de classe du projet ABter

Reprenons l'exemple de la séquence simple spécifiée dans le projet AB, que nous reprodusons, par commodité, sur la figure IV-43. Etant donné les restrictions que nous nous sommes imposées ci-dessus (Cf. IV.C.4.3.b), il nous reste à expliciter pour le projet un déploiement sur un seul espace d'adresse qui spécifie la démultiplication de l'objet `b` sous la forme d'une ferme d'objets. L'entité que nous allons annoter par le stéréotype «`farm`» ne peut donc être un nœud car nous ne démultiplions pas ici l'espace d'adresse. Il nous reste la possibilité d'annoter ou bien une instance de l'objet `b` sur un diagramme d'instance de déploiement ou bien un composant réalisant la classe `B` sur un diagramme de déploiement. Nous choisissons cette deuxième possibilité par esprit de continuité avec les exemples déjà exposés.

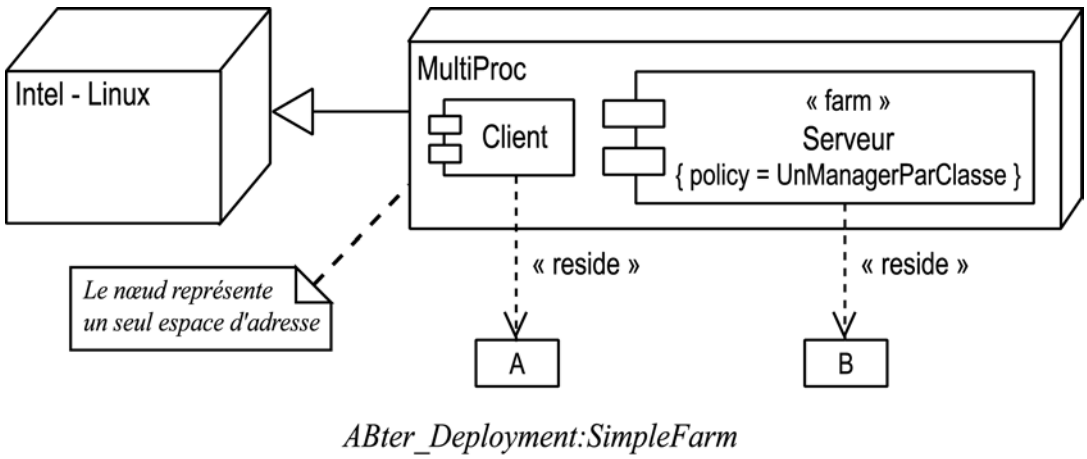


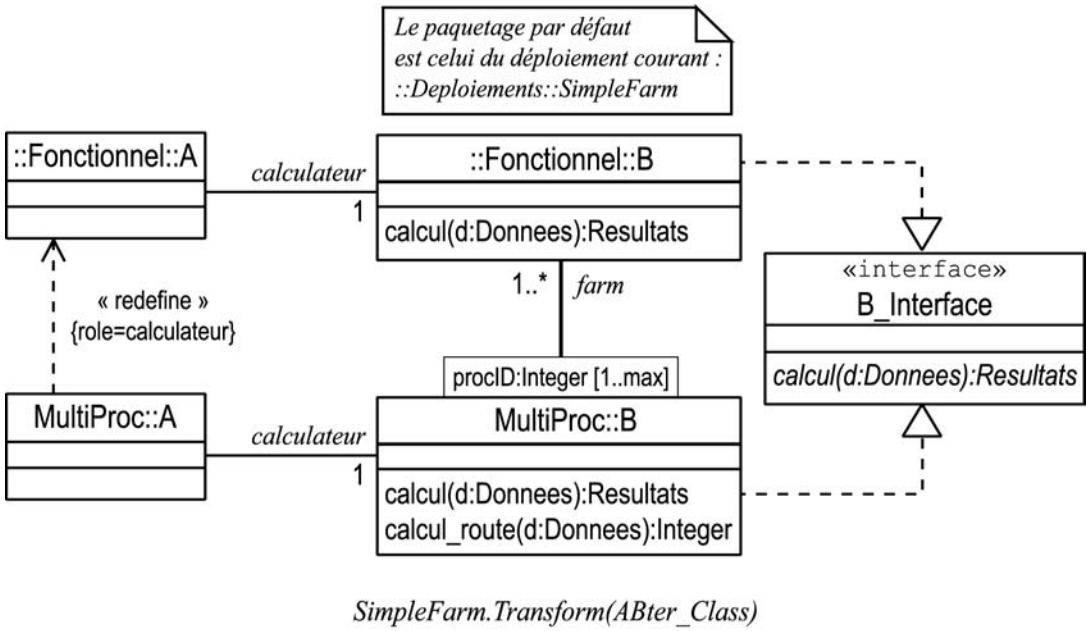
Fig. IV-44 : Diagramme de déploiement du projet ABter, spécifiant un déploiement en ferme

La figure IV-44 montre le diagramme du déploiement `SimpleFarm` renvoyant au motif `UnManagerParClasse` pour le déploiement en ferme du composant `Serveur` où réside l'objet `b`, sur le nœud `MultiProc` de type `Intel-Linux`. Les figures IV-45 et IV-46 montrent les résultats de la procédure de transformation

du modèle fonctionnel par ce motif. Celle-ci peut être décomposée comme suit :

1. Les « classes cibles » résidant sur un composant démultiplié sont déterminées à partir des diagrammes de déploiement. Dans notre exemple, seul la classe `B`, jouant le rôle de calculateur pour la classe `A` et résidant sur le composant démultiplié `Serveur`, est dans ce cas (fig. IV-43 et IV-44).

2. A partir chaque classe cible `C`, une interface $\texttt{I}_\texttt{C}$ incluant toutes ses opérations publiques est extraite. Dans notre exemple, l'interface `B_Interface` est extraite de la classe cible `B` (fig. IV-45).



SimpleFarm.Transform(ABter_Class)

Fig. IV-45 : Diagramme de classe du projet ABter transformé par le méta-module d'implémentation de fermes `UnManagerParClasse`

3. Pour chaque classe cible `C`, une classe « gestionnaire de ferme » `C'` réalisant $\texttt{I}_\texttt{C}$ est créée ; celle-ci est associée à `C` à travers le rôle `farm` de cardinalité multiple correspondant à la taille de la ferme et qualifé par la clé entière `procID`. Les méthodes de `C'` correspondant à l'interface $\texttt{I}_\texttt{C}$ sont destinées à router l'appel vers la méthode homonyme de l'un des objets de la ferme. Pour chaque méthode de `C'`réalisant l'une des opérations de $\texttt{I}_\texttt{C}$, la politique de routage est encapsulée dans une méthode associée dont le nom est formé du nom de la méthode concaténée au suffixe `_route` et prenant en entrée les mêmes paramètres et renvoyant une clé entière qualifiant l'objet de la ferme vers lequel l'appel doit être routé. Dans notre exemple (fig. IV-45), le gestionnaire de ferme est réalisé par la classe `MultiProc::B` associée à une ferme d'objets de classe `::Fonctionnel::B`. La seule méthode concernée est

`calcul(d:Donnees):Resultats` ; elle est associée à un algorithme de routage à travers la méthode `calcul_route(d:Donnees):Integer`.

4. Dans le paquetage correspondant au nœud de déploiement, toute classe appelant une classe cible est remplacée par une classe équivalente associée à la classe cible équivalente dans ce paquetage. Les caractéristiques (rôles, cardinalités,...) de ces associations équivalentes sont maintenues à l'identique. Dans notre exemple (fig. IV-45), la classe appelante `::Fonctionnel::A` associée à la classe `::Fonctionnel::B` donne lieu sur le nœud `MultiProc` à la classe `MultiProc::A` associée à la classe `MultiProc::B` (dépendance «`redefine`» liée au rôle calculateur de l'association entre `A` et `B`).

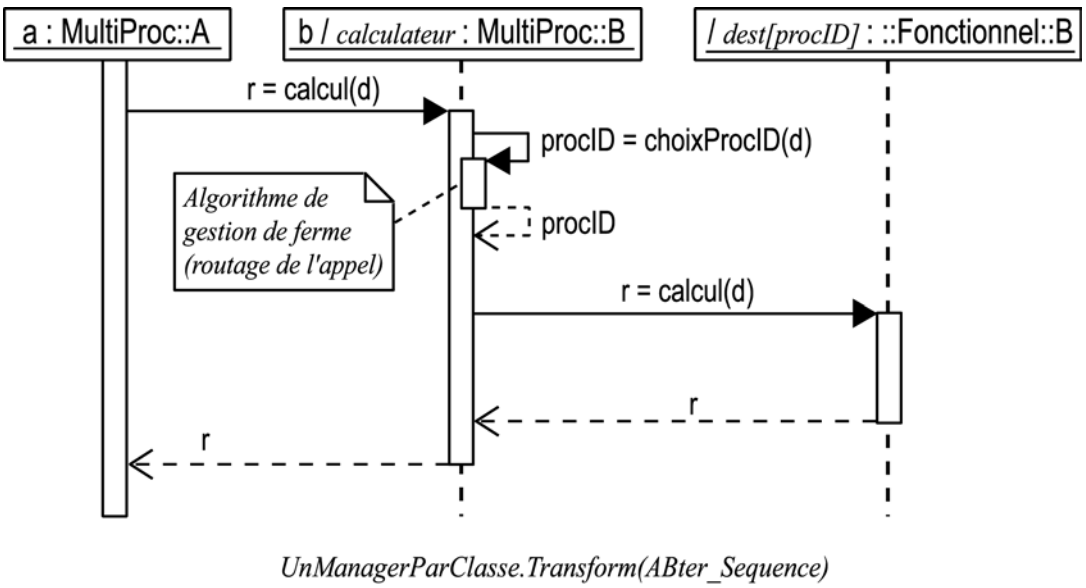


Fig. IV-46 : Séquences du projet ABter résultant du la transformation par le méta-module `UnManagerParClasse`. Ces séquences sont directement assimilable à du code C++ ou Java.

5. Les diagrammes de séquence effectifs directement interprétables pour la génération de code sont produits par le méta-module selon le protocole souhaité. Pour ce qui est de notre exemple, la figure IV-46 représente la séquence d'appel transparent, de routage et d'appel effectif de la méthode `calcul(Donnees):Resultats`. par l'objet `a` de classe `MultiProc::A`.

IV.C.4.4 Résumé des cyles de spécification/transformation

En nous appuyant sur une généralisation des trois exemples que nous venons d'étudier, nous pouvons représenter nos procédures de développement ainsi que leurs diagrammes et outils associés sous la forme du cycle de la figure IV-47. Ce cycle peut essentiellement se résumer en trois groupes de procédures principaux :

- Spécifications Fonctionnelles
- Spécifications de déploiement
- Génération des classes effectives

sachant qu'à un tel cycle succède un autre soit par raffinement et évolution du modèle fonctionnel, soit par redéploiement du même modèle fonctionnel.

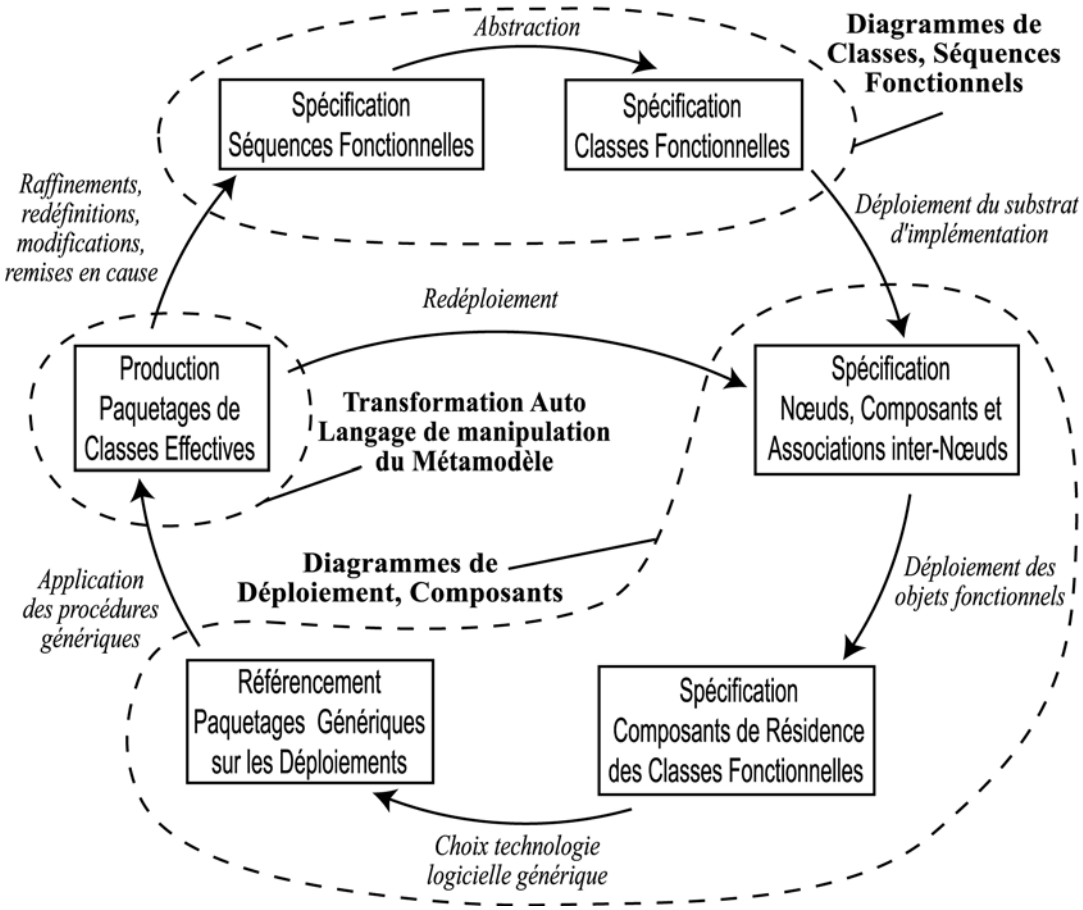


Fig. IV-47 : Cycle des procédures de développement ; diagrammes et outils associés

Il est également important de souligner que le groupe « Spécifications de déploiement » utilise les éléments de modélisation (classes, séquences,...) produits par le groupe « Spécifications Fonctionnelles » puisqu'il s'agit d'y spécifier dans quels composants et sur quels nœuds les classes fonctionnelles doivent résider, et que le groupe « Génération des classes effectives » utilise les éléments produits par les deux autres groupes, puisqu'il s'agit d'y générer les classes effectives en transformant les classes fonctionnelles et en produisant des classes d'implémentation à partir des topologies de déploiement et grâce aux procédures de transformation de modèle des modules génériques spécifiés dans les déploiements.

IV.C.5 Composition de transformations

Dans les exemples du IV.C.4.1, IV.C.4.2 et IV.C.4.3, nous avons traité sous forme de motifs récurrents particuliers correspondant à des développements réels, les problématiques d'appel de méthode distant, d'appel asynchrone et de déploiement en ferme. A chaque fois, le modèle traité est simpliste afin de nous en tenir strictement au motif étudié ainsi qu'aux procédures de spécification de déploiement et de transformation associées. Bien entendu, dans un système réel, ces motifs sont le plus souvent appelés à être composés entre eux ; en effet, il est probable qu'un système TDAQ réaliste mette en œuvre *en même temps* des motifs associés à ces trois problématiques. Nous avons vu par exemple que le système d'acquisition/déclenchement de l'expérience ANTARES est dans ce cas.

Il nous faut donc expliciter les contraintes et les lignes de conduite à adopter afin de permettre une spécification rigoureuse de la coexistence de trois motifs liés aux problématiques d'appel distant, d'appel asynchrone et de déploiement en ferme. Autrement dit, quels que soient les motifs particuliers mis en œuvre dans une application donnée, ceux-ci doivent pouvoir être composés les uns avec les autres sans ambiguïtés ni contradictions. Notre description doit recouvrir les 4 situations possibles correspondant à la composition de deux motifs (3 cas) et au cas où les 3 motifs sont composés. Par continuité et afin de faciliter les comparaisons avec les chapitres précédents, nous continuerons à utiliser les mêmes modèles fonctionnels qu'au IV.C.4.

IV.C.5.1 L'appel distant asynchrone

Ce premier cas correspond à la spécification et à la transformation de modèle résultant de la composition de l'appel distant et de l'appel asynchrone. Il s'agit, en d'autres termes, de mettre en œuvre de manière transparente l'appel asynchrone d'une méthode appartenant à un objet situé dans un espace d'adresse différent de celui de l'objet appelant.

Reprenons le modèle fonctionnel du projet ABbis de la figure IV-38 reproduit par commodité sur la figure IV-48. Ce modèle comprend l'appel asynchrone de la méthode `B::calcul()` par l'objet de classe `A`. A ce modèle fonctionnel, nous associons un nouveau déploiement représenté sur la figure IV-49.

Ce diagramme est pour l'essentiel identique au diagramme de déploiement de

la figure IV-33 utilisé pour l'exemple de l'appel distant synchrone. Nous y avons simplement ajouté la même spécification de motif d'appel asynchrone sur le nœud `Acquisition` que celle employée dans le diagramme de déploiement de la figure IV-37.

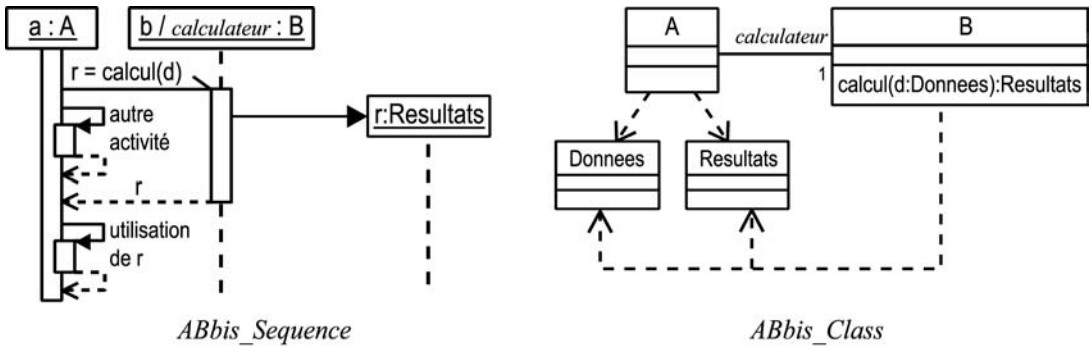


Fig. IV-48 : Diagrammes fonctionnels de séquence et de classe du projet ABbis

L'interprétation naturelle de ce diagramme implique, entre autres, que les appels asynchrones sont autorisés sur le nœud `Acquisition` et doivent être implémentés selon le motif `UneQueueParMethode`. Or, le modèle fonctionnel spécifie un appel asynchrone effectué par l'objet `a` (de classe `A`) sur une méthode de l'objet `b` (de classe `B`) jouant le rôle `calculateur` pour `a` ; le déploiement, de son côté, place les objets `a` et `b` sur deux nœuds différents (resp. `Acquisition` et `Traitement`). Par conséquent, les classes effectives produites par la composition des deux modules génériques `UneConnexionParMethode` et `UneQueueParMethode` doivent permettre l'appel asynchrone de la méthode de `b` par `a` à travers un appel distant.

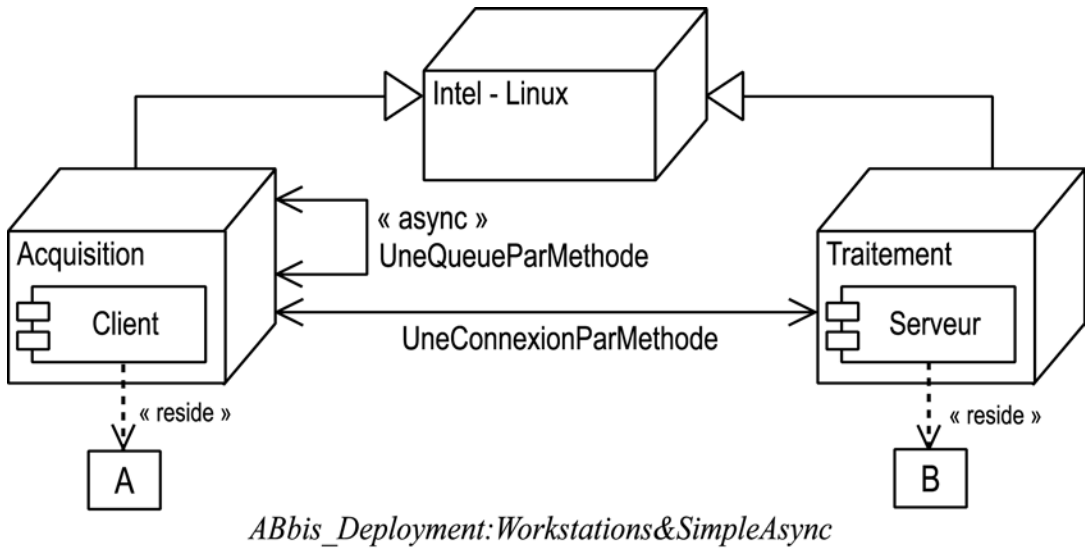


Fig. IV-49 : Déploiement du projet ABbis sur 2 espaces d'adresse dont un avec spécification d'appels asynchrones

En supposant les classes effectives implémentant l'appel distant produites dans chacun des 2 paquetages associés aux 2 nœuds du déploiement (Cf. IV.C.4.1.d ainsi que la figure IV-34), il existe *a priori* deux manières de mettre en œuvre l'asynchronisme spécifié dans le méta-module `UneQueueParMethode` : l'une implémenterait l'appel asynchrone de `::Fonctionnel::B::calcul` par `Traitement::B_Reception`, c'est-à-dire l'appel asynchrone de la méthode `calcul` de l'objet cible `b` par l'objet de réception de l'appel distant (dans la terminologie CORBA, le *skeleton*) ; l'autre placerait l'appel asynchrone au niveau de la méthode `Acquisition::B::calcul`, c'est-à-dire que l'objet `a` de classe `Acquisition::A` effectuerait un appel asynchrone de la méthode `calcul` de l'objet `Acquisition::B` jouant pour `a` le rôle `calculateur`. Cette deuxième implémentation est bien entendu plus naturelle car conforme à la spécification de déploiement de la figure IV-49 qui lie le métamodule d'appel asynchrone au nœud `Acquisition`.

D'une manière générale, la décision d'effectuer un appel de méthode de manière asynchrone relève de la nécessité de découpler deux processus dont les contraintes temporelles sont très différentes. Par exemple, le processus appelant est une interruption matérielle qui doit prendre fin très rapidement, sans attendre que l'appel qu'il déclenche sur un processus d'acquisition prenne fin. L'appel asynchrone complique la logique de l'application et il n'est donc introduit que si l'appelant doit impérativement « reprendre la main » sans courir le risque d'être bloqué par l'appelé. Or, de par sa nature, un appel distant est plus lent à s'effectuer qu'un appel dans le même espace d'adresse et son caractère éventuellement bloquant pour l'appelant en est de ce fait renforcé. Par conséquent, si le développeur estime que pour des raisons de découplage, tel appel doit être asynchrone, il est naturel de considérer que dans le cas où l'appel s'effectue à distance, l'asynchronisme doit être introduit au niveau de l'espace d'adresse appelant afin d'économiser sur les latences dues à la transmission de l'appel à travers les espaces d'adresses. Dans notre exemple, le déploiement de l'asynchronisme au niveau du nœud contenant l'objet appelant correspond donc au cas le plus fréquent. Aussi pensons-nous que la spécification des motifs de communication asynchrone à distance peut être notée sous une forme plus concise à condition d'imposer que l'interprétation de cette notation situe la mise en œuvre de l'asynchronisme au niveau du nœud appelant. En utilisant cette notation plus concise, notre diagramme de la

figure IV-49 peut-être remplacé par celui de la figure IV-50.

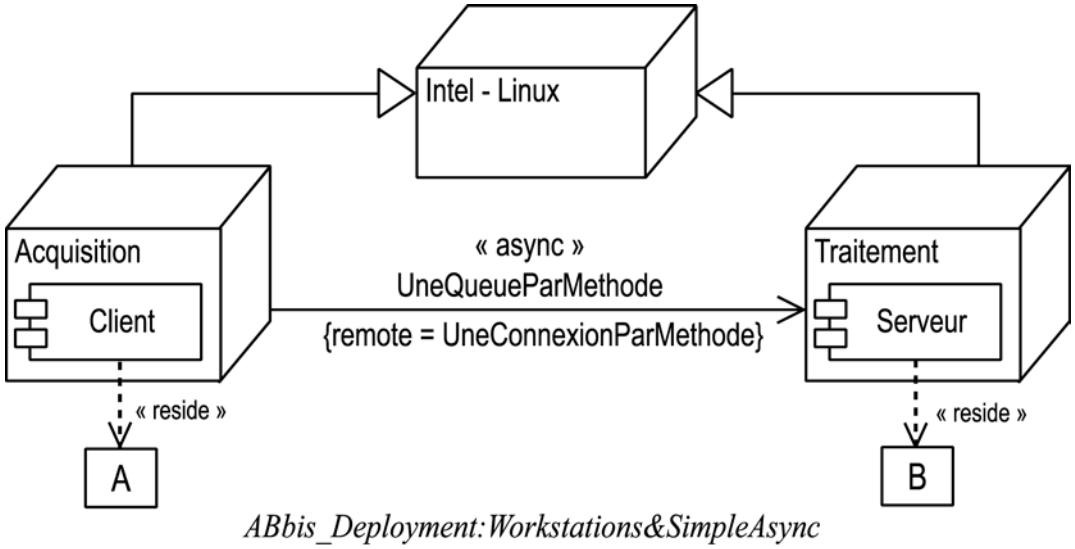


Fig. IV-50 : Déploiement du projet ABbis utilisant une notation réduite pour spécifier les méta-modules d'appels distants et asynchrones

Ainsi, en reliant deux nœuds sur un diagramme de déploiement par une association de type «`asynch`», on spécifie qu'il peut y avoir appel asynchrone d'un nœud à l'autre. Le nom de l'association renvoie au métamodule d'appel asynchrone et l'étiquette `remote` associée renvoie au métamodule d'appel distant, les deux modules étant indispensables pour pouvoir produire toutes les classes nécessaires à l'implémentation de l'appel d'une méthode de l'objet `b` par l'objet `a`. Insistons enfin sur le fait que la procédure de transformation de modèle liée à l'implémentation de l'appel asynchrone doit s'appliquer au niveau du nœud appelant, c'est-à-dire essentiellement au niveau de la communication entre l'objet appelant et le *« proxy »* de l'objet appelé. Une autre façon d'exprimer les choses est qu'il faut d'abord appliquer le motif de distribution sur plusieurs nœuds et ensuite seulement les éventuels motifs liés à un seul nœud (comme l'appel asynchrone).

IV.C.5.2 Le déploiement en ferme à distance

Ainsi que nous l'avions précisé au IV.C.4.3, nous avons considéré, pour des raison de factorisation conceptuelle, le déploiement en ferme d'un objet selon le modèle d'une démultiplication en plusieurs objets au sein du même espace d'adresse. Etant donné que l'intérêt pratique du déploiement en ferme dans les systèmes TDAQ réside essentiellement dans le gain de performances, ce modèle serait adapté à un déploiement sur une plate-forme multiprocesseurs où l'objet

démultiplié serait un objet *multi-thread*. Cependant, dans l'essentiel des applications TDAQ, le recours au déploiement en ferme se traduit par l'exécution des algorithme de traitement/acquisition sur une *ferme de stations en réseau*, ce qui implique en particulier que l'objet cible soit démultiplié non plus au sein du même espace d'adresse mais bien sur des processus (et donc des espace d'adresse) multiples. En termes UML, cela signifie que le déploiement en ferme se fait sur plusieurs nœuds.

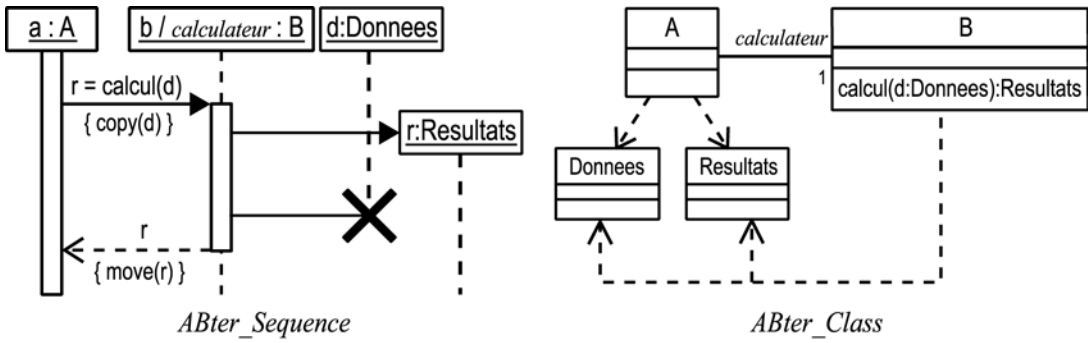


Fig. IV-51 : Spécification fonctionnelle du projet ABter

Reprenons la spécification fonctionnelle du projet ABter, reproduite par commodité sur la figure IV-51. Son déploiement en ferme pure (dans le même espace d'adresse), également reproduit en figure IV-52, est spécifié essentiellement grâce au stéréotype «`farm`» appliqué au composant `Serveur` où réside la classe de l'objet démultiplié, l'étiquette `policy` associée à ce stéréotype permettant au développeur de spécifier le méta-module mettant en œuvre le motif de déploiement en ferme de son choix (ici, le motif `UnManagerParClasse`, décrit en détail au IV.C.4.3.c). Comment pouvons-nous, de manière analogue, spécifier que le déploiement doit se faire en démultipliant un composant non plus au sein d'un même espace d'adresse mais à travers plusieurs nœuds ?

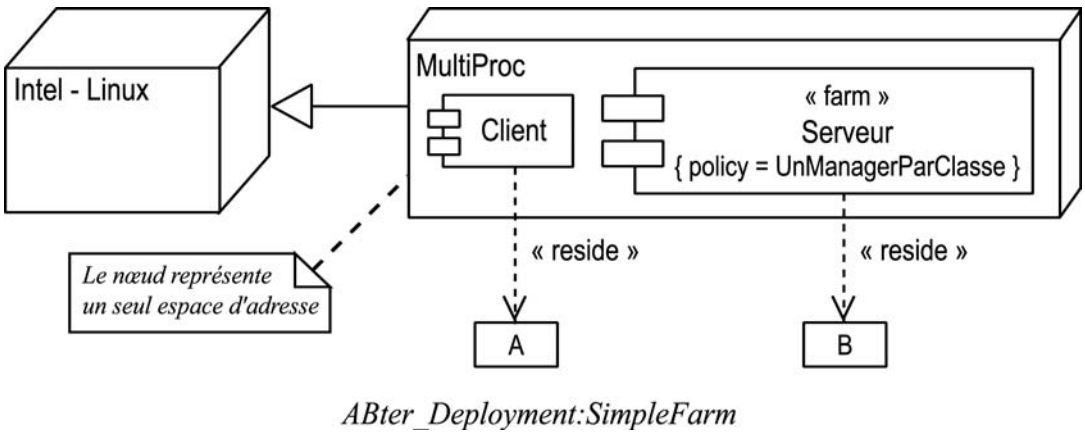


Fig. IV-52 : Déploiement en « ferme pure » du projet ABter

La solution la plus naturelle est bien entendu d'appliquer le stéréotype «`farm`» et son étiquette `policy` non plus à un seul composant mais au nœud. Il est également clair que l'objet appelant doive tourner sur un nœud distinct puisqu'il ne doit pas subir de démultiplication. La figure IV-53 montre ainsi un diagramme possible de déploiement en ferme à distance.

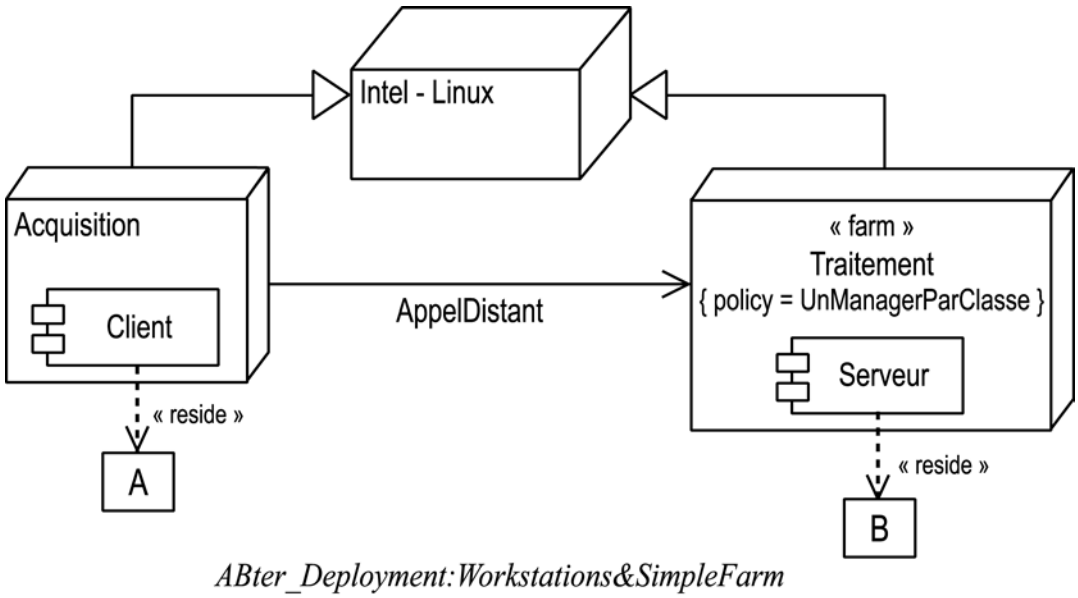


Fig. IV-53 : Redéploiement du projet ABter en une ferme de stations pour le composant Serveur

Il nous reste a définir comment les transformations du modèle fonctionnel défnies par les deux méta-modules d'appel distant et de déploiement en ferme doivent se composer afin de produire l'ensemble des paquetages de classes effectives. Une question, analogue à la problématique rencontrée au IV.C.5.1 se pose : sachant que le méta-module de déploiement en ferme produit une classe « gestionnaire de ferme » (Cf. IV.C.4.3.c, classe `MultiProc::B` de la figure IV-45) permettant l'accès transparent à l'objet démultiplié comme à un seul objet, des communications entre objet client et gestionnaire ou entre gestionnaire et objet cible, lesquels doivent faire l'objet d'une transformation par le méta-module d'appel distant ? La réponse naturelle est de placer le gestionnaire de ferme sur le nœud de l'objet client et d'appliquer ainsi le motif d'appel distant aux communications entre le gestionnaire et les objets cibles. En effet, toute autre solution conduirait à placer l'objet gestionnaire sur l'un de nœuds de la ferme ce qui reviendrait d'une part, à privilégier un nœud particulier sans raison particulière et d'autre part, à imposer une communication à distance *a priori* inutile entre l'objet client et le gestionnaire (puisqu'ils ne résideraient pas sur le même nœud). Pour reprendre les entités de notre exemple (fig. IV-53), l'objet gestionnaire

produit par le motif de déploiement en ferme sera placé sur le nœud `Acquisition` et transmettra ses appels aux objets cibles déployés sur *les* nœuds `Traitement` ; ainsi l'objet client `a` verra le gestionnaire de ferme, dans son propre espace d'adresse, comme un objet ayant la même interface que les objets cibles, rendant ainsi transparent l'accès à l'objet fonctionnel `b` déployé sur une ferme de processeurs distants.

IV.C.6 Autres annotations et spécifications

Dans nos trois exemples de transformation de modèle (IV.C.4.1, IV.C.4.2 et IV.C.4.3) l'articulation entre transformations génériques, modèle fonctionnel et modèles de déploiement nous a conduit à introduire un certain nombre de contraintes et de spécifications définissant notre processus de modélisation d'un système TDAQ. Quelques extensions UML ont du être proposées afin de pouvoir exprimer ces contraintes et spécifications de manière suffisamment précise pour pouvoir donner lieu à des transformations de modèle automatiques et, *in fine*, à de la génération automatique de code. Il demeure cependant quelques ambiguïtés que nous nous proposons de lever en précisant ou en explicitant les points qui vont suivre. Dans toutes nos discussions, afin de fixer les idées sans sacrifier à la généralité de notre propos, nous considérerons les classes `A` et `B` telles que présentées dans les modèles fonctionnels du IV.C.4.1, IV.C.4.2 et IV.C.4.3 (fig. IV-32, IV-38 et IV-43), où `A` est une classe appelante associée et une classe appelée `B` sous le rôle `calculateur`. Le nom du paquetage contenant le moèle fonctionnel sera « Fonctionnel », ce qui implique que la désignation complète des classes fonctionnelles `A` et `B` est `::Fonctionnel::A` et `::Fonctionnel::B`.

IV.C.6.1 Redéfinition d'une association dans le modèle transformé

Dans notre processus de modélisation, les transformations du modèle fonctionnel à partir de la spécification d'un modèle de déploiement aboutissent à la production d'un ou plusieurs sous-systèmes de classes effectives, chacun correspondant à l'un des nœuds introduits par le déploiement (Cf. par exemple les paquetages produits dans le cas de l'appel distant synchrone, fig. IV-36). Il s'agit à chaque fois de reproduire dans le sous-système associé à un nœud particulier une version *effective* (opérationnelle) des classes fonctionnelles et, plus particulièrement, de reproduire au niveau de ces classes effectives les mêmes rapports que ceux qui existent entre leurs homologues fonctionnels. Ainsi, un sous-sytème

`SubSys` correspondant à un nœud où réside la classe `A` comporte nécessairement une classe équivalente à `B` (proxy ou autre) associée à `A` puisque les deux classes sont associées dans le modèle fonctionnel. En d'autres termes, du fait d'une part, que `::Fonctionnel::A` soit associée à `::Fonctionnel::B` et que d'autre part, le modèle de déploiement spécifie que `A` réside sur le nœud `SubSys`, deux classes `SubSys::A` et `SubSys::B` doivent être créées. De plus, ces deux classes doivent entretenir les mêmes rapports que `::Fonctionnel::A` et `::Fonctionnel::B`. En même temps, nous voulons que `SubSys::A` hérite de toutes les spécifications (et tout le code) de `::Fonctionnel::A` qui ne sont pas atteintes par la transformation vers le paquetage `SubSys`. Cependant, nous ne pouvons recourir ici à une relation de spécialisation entre `SubSys::A` et `::Fonctionnel::A` car nous ne voulons pas que `SubSys::A` hérite de l'association entre `::Fonctionnel::A` et `::Fonctionnel::B`. En d'autres termes, `SubSys::A` doit être définie comme une spécialisation de `::Fonctionnel::A` *sauf* en ce qui concerne son association avec `::Fonctionnel::B` qui doit être remplacée par une association de mêmes caractéristiques avec `SubSys::B`. Nous avons besoin pour cela de pouvoir redéfinir une association tout en maintenant ses caractéristiques de la même manière que nous pouvons redéfinir une méthode tout en maintenant sa signature conforme à l'opération associée.

Si nous nous référons aux différents stéréotypes de dépendance de la norme UML ([B-12], pp. 2-17, 2-18), nous constatons que notre transformation de la classe `A` relève aussi bien du stéréotype `«refine»` (Cf. également [B-12], pp. 3-91, 3-92 et B-16) que du stéréotype `«realize»` . Elle tient également de la relation de *spécialisation* puisque tout ce qui n'est pas redéfini par la transformation doit être hérité tel quel (notamment : les méthodes de `A` ainsi que les attributs de `A` n'intervenant pas dans ses relations avec `B`,). Aussi, pour désigner cette forme particulière de redéfinition d'une classe, avons-nous défini un stéréotype propre : `«redefine»`, associé à une relation de dépendance et paramétrée par des étiquettes précisant les rôles dont les associations sont redéfinies au sein du modèle transformé. Nous pouvons ainsi spécifier précisément les associations redéfinies par une procédure de transformation de modèle (Exemple : fig. IV-45, p. IV-93).

IV.C.6.2 La question de l'accès distant à un attribut

Dans notre exemple de l'appel distant synchrone, les communications entre

objets distants s'effectuent uniquement par appel de méthode distant. Plus précisément, un appel de méthode tel l'appel de la méthode `calcul` de `b` par `a`, spécifié dans le diagramme de séquence du modèle fonctionnel, devient dans le modèle effectif un appel de méthode distant en raison du déploiement des classes `A` et `B` sur des nœuds différents. Que se passerait-il si, dans le modèle fonctionnel, l'objet `a` accédait non seulement à une méthode de `b` mais également à l'un de ses attributs publics ? Comment un tel accès doit-il être interprété en cas de déploiement des deux objets sur des nœuds différents ?

Il est clair que l'on peut, dans tout diagramme de déploiement, spécifier un méta-module d'accès distant aux attributs de manière analogue à la spécification de méta-modules d'appel de méthode à distance. La notation naturelle pour spécifier cela serait une association inter-nœuds spécialisée par un stéréotype (par exemple, «`access`»). Dans ce cas, le processus de transformation du modèle fonctionnel devra :

1. Déterminer, au sein de tous les diagrammes de séquence et d'activité fonctionnels liés à une classe, toutes les références à un attribut d'un rôle associé à cette classe, c'est-à-dire toutes les expression du type *<nomRole>.<nomAttribut>*.
2. Déterminer, pour ce déploiement particulier, si la classe qui remplit ce rôle réside sur un nœud différent de celui sur lequel réside la classe courante.
3. Générer, grâce au méta-module d'accès distant aux attributs spécifiés sur le déploiement, les classes effectives permettant l'accès transparent aux attributs des rôles au sein des diagrammes d'activité et de séquence.

Nous voyons donc, qu'il est possible, en principe, de traiter les cas où le développeur spécifierait l'accès à un attribut dans le modèle fonctionnel tout en spécifiant un déploiement tel que l'objet conteneur de cet attribut serait situé dans un espace d'adresse distant. Le traitement de ce cas relèverait, comme pour toutes les autres problématiques d'implémentation de mécanismes de distribution et de parallélisme, de la spécification de motifs d'implémentation idoines sous la forme de méta-modules référençables au sein du modèle de déploiement.[29]

Nous pensons cependant, qu'en ce qui concerne le développement de systèmes

29. On imagine aisément un motif mettant en œuvre des mécanisme de «cache» pour l'accès aux attributs à travers un objet «proxy».

TDAQ, la possibilité d'accès distant aux attributs d'un objet ou tout autre forme de communication inter-objet autre que l'appel de méthode n'est pas réellement nécessaire et qu'il serait probablement plus économique de restreindre la liberté d'action du développeur en la matière. Dans ce cas, il suffit à notre canevas de développement de générer un message d'erreur circonstancié dans les cas où un accès distant à un attribut est détecté au sein d'un modèle de déploiement.

IV.D DU CANEVAS MÉTHODOLOGIQUE AU CANEVAS CLASSIQUE

Nous avons vu dans les deux sections précédentes notre proposition de cycle de développement pour systèmes TDAQ (IV.B) et le processus de modélisation associé (IV.C). Les principes sur lesquels reposent ces deux grands volets de notre travail permettent, ainsi qu'annoncé au III.C.5, de mettre en place ce que nous avons appelé un « canevas méthodologique », c'est-à-dire un cadre de développement de systèmes TDAQ qui, avant tout, oriente et balise le travail des développeurs de telle manière que les modules développés soient réutilisables et/ou possèdent des qualités de généricité. La séparation entre les aspects intrinsèques à l'application et les aspects liés aux contraintes d'implémentation, et ce, aussi bien dans la décomposition du cycle de développement que dans l'organisation des paquetages, incite le développeur à distinguer dès le départ les modules non réutilisables des modules réutilisables. En effet, les premiers sont spécifiques à l'application alors que les seconds répondent à un problème de *projection* de l'application sur une plateforme d'implémentation. Dans ces derniers nous retrouverons d'ailleurs beaucoup de *méta-modules* car les problématiques d'implémentation se traduisent le plus souvent par une problématique de transformation du modèle fonctionnel en modèle effectif selon un motif récurrent (Cf. la discussion sur la formalisation des motifs récurrents au III.B.4.3). Or, la variabilité des motifs récurrents et, plus généralement, des problématiques d'implémentation est inférieure à la variabilité des spécificités applicatives elles-mêmes[30]. Aussi, est-il raisonnable d'espérer que le développement de méta-modules ou de modules d'implémentation totalement nouveaux au cours de la vie du canevas devienne de moins en moins nécessaire, le développeur pouvant de plus en plus recourir à une réutilisation des modules déjà développés. Par exemple, il ne peut y avoir profusion de

30. Même si les applications connaissent également des motifs invariants de haut niveau tel l'assemblage d'événements (Cf. II.A.3)*

méta-modules d'appel distant car les mêmes concepts se retrouvent toujours, éventuellement mis en œuvre d'une manière adaptée au type d'implémentation recherché (entre matériel et logiciel, entre stations sur réseau standard, etc.). Une autre façon de dire les choses est que si *a priori*, les modèles de déploiement particuliers possibles sont très nombreux, les *types* de déploiement, c'est-à-dire les grandes classes de modèles de déploiement sont en nombre restreints car les plate-formes de traitement et de communication sont en nombre fini. Nous ne pensons pas cependant que ces types puissent être définis par avance à partir d'une simple analyse de domaine : *ce sont sont les développements eux-mêmes, effectués dans le cadre de notre canevas méthodologique, qui mettront progressivement au jour ces classes de modèles de déploiement des systèmes TDAQ*.

On peut ainsi voir que ce qui est au départ un canevas méthodologique essentiellement réduit à un procédé de développement et à un processus de modélisation évoluera au cours des développements d'applications réelles vers un canevas plus proche de la notion de canevas classique au sens où le développeur aura de plus en plus accès à des menus de modules et méta-modules réutilisables. Au fur et à mesure de l'usage du canevas méthodologique pour la conception et la réalisation d'applications réelles, les bibliothèques de modules et méta-modules d'implémentation s'enrichissent, permettant au développeur de s'en tenir à la conception des modèles fonctionnels et des modèles de déploiement, la projection des premiers sur les seconds devenant de plus en plus automatique.

V

CONCLUSION ET TRAVAUX FUTURS

Nous avons détaillé en section II les caractéristiques et évolutions des systèmes TDAQ et nous avons conclu à la nécessité pour les laboratoires participant aux développements de tels systèmes de se doter d'outils conceptuels et méthodologiques mettant en œuvre une forme de réutilisation logicielle afin de pouvoir faire face efficacement à la croissance des systèmes présents et futurs et ce, en dépit de la diversité des systèmes à développer. Nous avons ensuite décrit en section III les technologies sur lesquelles nous comptions fonder notre proposition en la matière, notamment celles des motifs récurrents et des canevas. Nous avons conclu à la nécessité de recourir à une forme de canevas que nous avons qualifiée de « méthodologique », différente des canevas métier classiques par son intégration avec un processus de développement et de modélisation spécifiques. En section IV, cœur de notre travail de thèse, nous avons détaillé notre proposition originale en explicitant d'abord au IV.B un *cycle de développement* spécialement adapté aux problématiques rencontrées dans le développement des systèmes TDAQ et ensuite, au IV.C, un *processus de modélisation* associé au cycle de développement, s'appuyant sur la notation UML et ses possibilités d'extension, et s'articulant autour du triptyque [Modèle fonctionnel – Modèle de déploiement – Motifs récurrents]. L'idée maîtresse de notre processus de modélisation est de traduire concrètement le caractère incrémental et « en spirale » du cycle de développement par la notion de « redéploiements » d'un même modèle fonctionnel sur des plate-formes multiples. A cet égard, nous faisons un usage *actif* des spécifications et diagrammes de déploiement UML (au-delà de ce que spécifie la norme [B-12] pp. 3-172 à 3-178) puisque ceux-ci constituent le lien entre le modèle fonctionnel et sa transformation par l'application des motifs récurrents pour l'obtention d'un modèle exécutable. La séparation explicite entre modèle fonctionnel et modèles de déploiements, associée à des contraintes de modélisation spécifiques à chaque classe de modèles (Cf. IV.C.3) permet de canaliser le travail du concepteur/développeur dans le sens d'une modularité plus stricte et donc d'une réutilisabilité améliorée des composants développés. L'explicitation des technologies d'implémentation sous la forme de motifs récurrents programmés dans des modules génériques séparés des applications permet leur réutilisabilité directe pour le développement d'autres systèmes. Enfin, en interdisant le mélange entre modèles strictement applicatifs (fonctionnels) et modèles d'implémentation, cette organisation du processus de développement minimise les

réécritures de code et les re-modélisations au cours du développement d'une même application, contribuant ainsi à une meilleure productivité de l'activité de développement.

V.A INTERACTIONS AVEC OMG* MDA*

Les conclusions et le modèle de développement auxquels nous avons abouti dans notre travail sur la rationalisation des développement de systèmes TDAQ rejoignent les réflexions récentes de l'OMG* sur l'ingénierie logicielle orientée objets, et ce, bien que les motivations premières soient quelque peu différentes au départ. Notre problématique concerne avant tout l'amélioration des processus de développement alors que les préoccupations de l'OMG sont essentiellement centrées sur l'interopérabilité entre technologies et outils informatiques. Cependant, malgré ces différences d'objectifs, les solutions envisagées présentent des similitudes conceptuelles suffisamment profondes pour que nous puissions envisager de fonder notre modèle de développement de systèmes TDAQ, au-delà de la seule notation UML*, sur le nouveau paradigme de modélisation de l'OMG: MDA (*Model Driven Architecture*) ou « Architecture Orientée Modèles ».

V.A.1 Cadre de modélisation général pour les domaines OMG

Parmi les activités de normalisation de l'OMG, les spécifications de domaines tiennent une place importante au sein des développements liés à l'architecture d'objets distribués CORBA* (Cf. page des groupes de travail de l'OMG à l'adresse http://www.omg.org/homepages/index.htm). Les spécifications de domaines consistent en des spécifications de composants et d'architectures spécifiques à un domaine particulier correspondant à un marché vertical tel la finance, les télécoms, le commerce électronique, etc. Ces spécifications sont originellement destinées à définir des canevas métier de développement d'applications distribuées fondés sur CORBA, aussi sont-elles essentiellement exprimées sous la forme d'interfaces de composants spécialisés dans le langage de description d'interfaces IDL (*Interface Description Language*, norme ISO/IEC 14750, Cf. [B-34]) dont la sémantique de modélisation est principalement explicitée sous la forme d'annotations textuelles. Or, ainsi que remarqué dans [B-32] p. 22, un service ou canevas bien conçu se fonde toujours sur un modèle sémantique (plus ou moins explicite) *a priori* indépendant de sa plateforme de mise en œuvre. Mais l'expression des spécifications

de domaines de l'OMG est aujourd'hui trop fortement liée à la plateforme CORBA alors qu'elles correspondent en réalité à des modèles de canevas et de services indépendants de toute plateforme particulière. Devant la prolifération des *middlewares** industriels de développement distribué (Cf. [B-33] p. 2), l'OMG a été conduit à un travail de réexpression de ces spécifications de domaines dans le cadre des *middlewares* les plus répandus. Cet état de fait a entraîné une prise de conscience sur la nécessité de recourir à des spécifications sous forme de modèles indépendants des plateformes de mise en œuvre, associées à des spécifications de *projection* de ces modèles sur des plateformes particulières. Ces plateformes étant elles-même des *méta-modèles* permettant une modélisation des applications selon des *points de vues* particuliers, la projection d'un modèle général sur une plateforme particulière est équivalente à la *transformation* d'un modèle appartenant à un méta-modèle source (par exemple UML) en un modèle cible appartenant à un autre méta-modèle (par exemple CORBA) (Cf. [C-42], plus particulièrement la section 6).

V.A.2 Qu'est-ce que le MDA ?

L'idée maîtresse du MDA est donc la vision de tout système comme une hiérarchie de points de vue correspondant à autant de modèles (Cf. [B-32], p. 4). Les modèles appartenant aux régions hautes de la hiérarchie correspondent aux points de vues de haut niveau indépendants des spécificités de telle ou telle plate-forme de déploiement : ce sont les PIM (*Platform Independent Model*) (Cf. [B-32], p. 9). Les modèles appartenant aux régions basses de la hiérarchie expriment les points de vue liés aux plate-formes d'implémentation particulières : ce sont les PSM (*Platform Specific Model*) (Cf. [B-32], p. 10). A cet égard, un code source peut être considéré comme un modèle situé relativement bas dans cette hiérarchie, le langage de programmation constituant alors le méta-modèle d'expression de ce modèle. La transformation de modèle permet, entre autre, le passage d'un modèle de haut niveau (PIM) à plusieurs modèles de bas niveau (PSM) selon les paramètres d'implémentation choisis par le concepteur/développeur d'une application (Cf. [B-32], p. 12). La génération de code, par exemple, correspond à une telle transformation, le paramètre principal étant dans ce cas le langage de programmation choisi[1].

Cette idée d'une cascade de modèles de plus en plus proches des détails

d'implémentation au fur et à mesure que l'on descend dans la hiérarchie permet également de définir la notion de « zoom » au sein d'un modèle ; [B-32] présente ainsi p. 5 l'exemple d'un protocole d'interaction défini à des niveaux de détails différents : dans la vision de haut niveau, l'objet A envoie un message à l'objet B. Un zoom sur cette interaction nous mène à un modèle plus détaillé où l'on voit que l'envoi de ce message correspond en fait à l'échange de plusieurs sous-messages correspondant à la mise en œuvre d'un protocole particulier, comprenant par exemple des accusés de réception, etc.

Notons enfin que dans la vision MDA de la hiérarchie de modèles d'un système, les niveaux bas sont globalement plus peuplés que les niveaux hauts car les alternatives d'implémentation d'un même modèle abstrait sont en général multiples, alors que la vision « quintessentielle » (de haut niveau) d'un système se résume à un nombre restreint de modèles possibles. Cette orientation est précisée formellement dans [B-32], figure 8, par les cardinalités du diagramme de classes décrivant le méta-modèle MDA.

V.A.3 Adéquation de notre canevas méthodologique au MDA

Les mécanismes de modélisation dégagés par le MDA rappellent clairement notre processus de modélisation par redéploiements. Il semble en effet assez clair que celui-ci s'inscrit naturellement dans l'architecture MDA puisque nous pouvons considérer notre « modèle fonctionnel » tel que défini au IV.C.3 comme un PIM et ses différentes réalisations sur des déploiements différents comme autant de PSM. A l'instar du MDA, nous prônons une spécification précise, voire formelle, des transformations permettant de passer du PIM aux PSM. Celles-ci correspondent, dans notre canevas, à la formalisation de motifs récurrents correspondant à des règles de bonne programmation ou à des technologies particulières. Pour reprendre les termes du MDA, nous pouvons dire que le méta-modèle de notre PIM est constitué par les diagrammes structurels et comportementaux d'UML ainsi que les métaclasses associées, étendus à des notions indispensables aux systèmes TDAQ tels les spécifications de passage de paramètre (Cf. IV.C.3.2), alors que le méta-modèle de nos PSM est constitué par les diagrammes de déploiement UML

1. *La transformation de modèle concerne également la progression entre PIMs, par exemple le passage d'un modèle d'analyse à un modèle de conception, ce qui correspond à un processus de raffinement ou d'enrichissement d'un modèle source en un modèle cible, sans que ce dernier soit lié à une plate-forme d'implémentation particulière.*

étendus et métaclasses associées ainsi que le langage de spécification des motifs récurrents[2] employé dans la définition des méta-modules génériques (Cf. IV.C.2.3).

Au premier abord, notre canevas TDAQ semble être restreint à une hiérarchie PIM → PSM simple à deux niveaux. En réalité, les niveaux sont sont susceptibles d'être plus nombreux à partir du moment où le déploiement devient suffisamment complexe pour faire intervenir une composition de méta-modules (Cf. IV.C.5), auquel cas nous sommes confrontés à l'équivalent de plusieurs niveaux de PSM. Par exemple, si nous considérons un modèle fonctionnel (PIM) que nous implémentons par l'intermédiaire d'un modèle de déploiement qui fait intervenir un méta-module d'appel distant et un méta-module de déploiement en ferme, nous obtenons une hiérarchie à trois niveau produite par deux tranformations :

Comme précisé par la spécification [B-32], p. 13, ce type de schéma est explicitement prévu par le MDA. En particulier, les transformations PSM → PSM sont destinées aux raffinements du modèle de déploiement, ce qui correspond exactement à l'usage que nous en faisons.

Il semble donc que les principes mis en œuvre dans notre canevas méthodologique révèlent une grande compatibilité avec la vision mise en avant par l'OMG à travers le MDA, à savoir, le développement « orienté modèles » des systèmes. Nous avons donc tout à gagner en termes de normalisation et d'interopérabilité avec de futurs outils de l'industrie à nous conformer aux recommandations existantes mais surtout à venir de l'OMG en la matière.

V.B PROJET DE MISE EN ŒUVRE DE NOTRE PROPOSITION

Nous avons commencé à appliquer les principes définis dans ce travail de thèse au projet concret que constitue le système d'acquisition offshore de l'expérience ANTARES ; nous avons également testé rétroactivement et à titre expériemental certains de ces principes sur quelques éléments du système de déclenchement des chambres à dérive de l'expérience NA48. Mais ces applications ont toutes été effectuées pour ainsi dire « à la main », c'est-à-dire, certes, en suivant avec précision les procédures décrites au IV.C, mais d'une manière totalement artisanale

2. Par exemple, le langage J dans le cas de l'AGL Objecteering (Cf. [A-28]).

et non automatisée. Ces essais nous permettent d'avoir bon espoir dans la cohérence et la pertinence de notre processus de modélisation mais celui-ci ne pourra révéler sa véritable utilité (notamment en termes de gains de productivité) que lorsqu'il sera mis en œuvre sur un AGL permettant l'automatisation des procédures de transformation de modèle et donc doté d'un langage de programmation au niveau du méta-modèle UML. Cette mise en œuvre à débuté sous la forme du projet de R&D MORDICUS* mené par le DAPNIA/SEDI, laboratoire d'accueil de cette thèse. Une brève description de ce projet est donnée en annexe au VI.E.

D'après notre analyse du V.A, il semble assez naturel de conformer notre canevas méthodologique aux normalisations de l'OMG dans le cadre du MDA. Nous avons déjà choisi UML comme langage de modélisation, il nous reste à nous assurer de demeurer conformes aux normalisations du MDA (qui ne font que commencer) pour la mise en œuvre du projet MORDICUS. Il nous faudra notamment prendre soin à ce que nos procédures de transformations respectent les recommandations du *Meta Object Facility* (MOF, Cf. [B-35]) qui constitue le méta-méta-modèle de l'OMG pour l'expression et la spécification des méta-modèles et des transformations de modèles. Il nous faudra également baser nos extensions concernant les questions de parallélisme et, plus généralement, de caractéristiques « temps réel », sur les profils UML idoines au fur et à mesure que ceux-ci seront définis par l'OMG et intégrés au cœur de l'architecture MDA (Cf. [B-32], p. 20). Enfin, il demeure même envisageable, à plus long terme, de définir au sein de la communauté HEP*, un profil de domaine spécifique aux applications TDAQ, voire de faire officialiser un tel profil par l'OMG. La création d'un tel profil serait extrêmement profitable à l'ensemble de développements lourds de la physique des hautes énergies en permettant une rationalisation des accrue des processus de développement et une interopérabilité précieuse entre sous-systèmes développés au sein des collaborations. Un tel objectif ne peut cependant être atteint que s'il y a un engagement clair de la communauté en ce sens, avec, notamment, une implication forte des institutions liées aux grands instruments telles le CERN ou FermiLab.

VI

ANNEXES

Cette section est dévolue à la description plus détaillée de questions et d'exemples abordés dans le reste du document, mais dont une présentation exhaustive au sein de l'exposé principal n'a pas été jugée utile. Toutes les sous-sections ci-après font l'objet d'au moins une référence dans le corps de la thèse. Il s'y trouve notamment, une partie sur le caractère « temps réel » des systèmes TDAQ*, ainsi que les différents sens que peut revêtir ce qualificatif au sein de ces systèmes (VI.A). Une seconde partie apporte quelques précisions sur les notions d'efficacité et de réjection des sous-systèmes de déclenchement.

VI.A CONTRAINTES TEMPS RÉEL DES SYSTÈMES TDAQ

Comme décrit au II.A.1 et sur la figure II-1, tout sous-système TDAQ est structuré sur le modèle d'une succession plus ou moins longue d'étages de déclenchement (typiquement 2 ou trois étages). A chaque niveau de déclenchement n, la capacité limitée des mémoires tampon du système d'acquisition impose une latence stricte ou statistique (notée L_n) au delà de laquelle la décision d'acquisition devra avoir été prise afin que les données anciennes laissent la place à des données plus récentes. L_n est donc le *budget temporel* dont disposent les sous-systèmes de déclenchement et d'acquisition au niveau n, l'un pour effectuer des traitements algorithmiques fondés sur la physique de l'expérience afin de sélectionner parmi les données celles qui doivent être acceptées, et l'autre pour transférer les données acceptées au niveau suivant avant que sa mémoire tampon ne soit pleine. Notons $L^T_n(e)$ le temps consommé par le sous-système de déclenchement pour traiter l'événement e et $L^A_n(e)$ le temps mis par le sous-système d'acquisition pour transmettre les données de l'événement au niveau suivant ($n+1$).

Le budget temporel L_n peut être fixe, ce qui sera par exemple le cas si les mémoires tampons de l'acquisition sont des mémoires « circulaires », c'est-à-dire adressées par la valeur courante de l'horloge de l'expérience. Dans ce cas, pour qu'il n'y ait pas faute temporelle, on doit avoir pour tout événement e : $L^T_n(e) + L^A_n(e) < L_n$. Ainsi, dans l'expérience NA48, les mémoires circulaires du premier niveau de l'acquisition peuvent contenir l'équivalent de 200 µs de données. La moitié de ce budget temporel étant allouée aux opérations d'acquisition proprement dites ($L^A_n(e) < 100$ µs), il reste 100 µs au système de déclenchement pour produire la décision d'acquisition, ce qui est une contrainte forte

étant donnée la complexité des algorithmes de sélection mis en œuvre [B-9] [C-1] [C-2] [C-3].

L_n peut également représenter une valeur moyenne. En effet, si la désallocation de la mémoire contenant les données d'un événement au niveau n ne survient pas au bout d'un temps fixe, alors L_n représente le budget temporel dont le déclenchement et l'acquisition devront se contenter *en moyenne* afin qu'il n'y ait pas dépassement de capacité dans les mémoires tampon du niveau n : $<L^T_n(e) + L^A_n(e)> < L_n$. Le système d'acquisition / déclenchement de niveau 2 de l'expérience ATLAS est un exemple d'une telle politique temps réel [C-4].

La contrainte temporelle sur les systèmes TDAQ n'est pas strictement de type « temps réel dur » au sens où, le système n'étant pas critique, les fautes temporelles sont permises. Par exemple, lorsque L_n est une valeur fixe, il se peut que pour certains événements, on ait dépassement de la latence allouée au sous-système de déclenchement, auquel cas, l'événement sera considéré comme « non traité » et se verra affecté d'une politique d'acquisition par défaut : acceptation ou rejet systématique ou encore acceptation variable selon une distribution donnée (par exemple, un événement non traité sur dix sera accepté). Cependant, la qualité des données voire la viabilité de l'expérience peuvent être remises en cause si le taux de fautes temporelles est tel que l'efficacité du déclenchement s'en trouve affectée. En effet, si le nombre de fautes temporelles dépasse un certain seuil[1], la perte d'événements physiques devient non négligeable et peut donc remettre en cause le principe de l'expérience. Quand aux fautes temporelles dans un sous-système d'acquisition, ils aboutissent à une perte brute de données et doivent par conséquent demeurer en deçà d'un certain seuil pour les mêmes raisons.

VI.B Pouvoir de réjection et efficacité

Le « pouvoir de réjection » et l'« efficacité » sont les deux caractéristiques quantitatives qui déterminent la qualité d'un sous-système de déclenchement*.

On définit le pouvoir de réjection ρ d'un système de déclenchement (ou de l'un de ses niveaux) comme le rapport entre le nombre d'événements rejetés et le nombre d'événements analysés ($\rho \in [0,1]$). D'autre part, les limitations intrinsèques des détecteurs, des algorithmes, des processeurs, etc., aboutissent à la

1. *Seuil déterminé le plus souvent heuristiquement. Voir section IV.*

réjection par erreur d'événements intéressants. Un paramètre important d'un système de déclenchement est donc également son efficacité ε, définie comme le rapport entre le nombre d'événements intéressants effectivement acceptés et le nombre d'événements intéressant total[2] ($\varepsilon \in [0,1]$). La qualité d'un système de déclenchement est donc une fonction croissante de ρ et ε, sachant que dans les conditions d'utilisation nominales du système, ρ et ε seront anticorrélés. L'optimisation du système consistera donc à régler ces deux paramètres non indépendants de manière à maximiser l'abondance de signal utile.

VI.C EXEMPLES D'HÉTÉROGÉNÉITÉ DES SOLUTIONS

Les spécificités propres à chaque sous-système TDAQ ainsi que le travail en commun d'équipes d'origines différentes entraînent naturellement une hétérogénéité importante entre les différentes technologies employées.

L'exemple des sous-systèmes de déclenchement et d'acquisition des chambres à dérive de l'expérience NA48 est, à ce titre, éloquente. Le traitement d'un événement par le sous-système de déclenchement des chambres à dérive. inclut l'exécution d'un algorithme complexe[3] en moins de 100 μs, le taux d'événements en entrée du système étant de 150 kHz. *A contrario*, le système d'acquisition qui lui est associé doit, d'une part, stocker dans des mémoires circulaires des événements au rythme d'environ 1 MHz, puis transmettre un sous-ensemble de ceux-ci en moins de 100 μs et à un taux d'environ 20 kHz. Le sous-système d'acquisition n'a donc pas du tout les mêmes besoins que le sous-système de déclenchement, ce dernier étant, par exemple, demandeur de bien plus de puissance processeur alors que le premier gère un flux de données plus important. On trouve ainsi, au sein du même détecteur, des sous-systèmes au caractéristiques très différentes, ce qui entraîne la coexistence de systèmes hétérogènes pourtant en interaction les uns avec les autres. A ce facteur d'hétérogénéité s'ajoute le fait que le sous-système de déclenchement a été développé par le DAPNIA/SEDI, le sous-système d'acquisition de premier niveau par l'Institut Siegen en Allemagne, et le sous-système d'acquisition de deuxième niveau par le CERN*. C'est ainsi que, *in fine*, ces trois sous-systèmes se retrouvent avec des systèmes d'exploitation aussi variés que Linux, SunOS, LynxOS, OS-9, sur des plate-formes à base de processeurs Intel,

2. *On définit l'« inefficacité » ι comme le nombre d'événements intéressant rejetés sur le nombre total d'événements intéressants. Par conséquent :* $\iota = 1 - \varepsilon$.
3. *Calculs de trajectoire, de point de désintégration, de moment transverse et de masse invariante.*

PowerPC, Sparc, ainsi que des DSP TMS320C40 de Texas Instruments et 96000 de Motorola, sans compter les modules de calcul spécialisés à base de FPGA nécessités par des latences maximales hors de portée des processeurs de l'industrie à l'époque de la conception du système [B-9] [C-1].

Dans l'expérience ANTARES, le sous-système d'acquisition offshore connaît des contraintes de consommation et de fiabilités très fortes évoquées en II.C.2, contraintes inexistantes pour les sous-systèmes d'acquisition et de déclenchement de plus haut niveau situés à terre. A cela s'ajoute le fait que le premier est développé par le DAPNIA/SEDI à Saclay alors que la responsabilité des seconds incombe à l'institut de physique des particules NIKHEF à Amsterdam, ce qui a tendance à entraîner des hétérogénéités dues à des cultures technologiques différentes. La conjugaison de ces deux facteurs produit des hétérogénéités qui semblent irréductibles, et ce, en dépit des efforts non négligeables d'unification des technologies employées. Ainsi les processeurs d'acquisition en mer sont des PowerPC basse consommation intégrant des ports de communication standards sur une seule puce et fonctionnant sous le système d'exploitation temps réel VxWorks [C-5], alors que les stations de calcul et d'acquisition à terre sont basées sur des plate-formes Linux tournant sur des processeurs Intel Pentium [B-14] [B-10].

VI.D Exemple d'évolution imprévue dans une expérience

Etant donné, d'une part, que la recherche fondamentale navigue souvent en des eaux inconnues et, d'autre part, que la complexité des expériences peut cacher certains paramètres importants, il arrive que les estimations de performances ne soient pas à la hauteur des contraintes effectivement rencontrées lors de l'expérience réelle : des taux d'événements peuvent se révéler plus élevés que ce que les simulations avaient prédit, une structure particulière imprévue peut apparaître dans la distribution d'un faisceau de particules primaires, etc. Dans ces cas, la solution se trouve dans une amélioration partielle d'un ou plusieurs sous-systèmes du détecteur. Par exemple, l'expérience NA48 a été confrontée à un taux moyen d'événements beaucoup plus élevé que prévu, ce qui a induit, entre autres, un fonctionnement du sous-système de déclenchement des chambres à dérive à la limite de ses capacités et donc des pertes de données dues aux fluctuations de taux d'événements. L'expérience n'a ainsi pu atteindre son régime de croisière qu'après

une amélioration de la ferme de processeurs temps réel [B-9]. Notons au passage que cette amélioration a également permis d'effectuer toute une classe de calculs plus complexes à la suite de nouvelles idées d'études physiques.

VI.E LE PROJET MORDICUS

Le projet MORDICUS* (Méthode à Objets Répartis de Développement Incrémental et de Conception UML* de Systèmes), inité par le DAPNIA/SEDI* est destiné à concrétiser les canevas méthodologique TDAQ* défini dans ce travail de thèse. Nous présentons ci-après une version légèrement modifiée de la proposition de projet telle qu'elle a été acceptée par le Conseil Scientifique et Technique de Service du SEDI.

VI.E.1 Objet du projet

L'objet du projet MORDICUS se décline en deux volets principaux :

1. Étude, élaboration et réalisation d'une méthodologie de conception / développement orientée objet pour les systèmes d'acquisition/traitement temps réel massivement distribués tels ceux développés par le DAPNIA pour les expériences de physique.
2. Développement d'un outil logiciel basé sur la notation standard UML mettant en œuvre cette méthodologie par la formalisation du processus de développement et par l'automatisation des motifs de conception récurrents.

VI.E.2 Problématique

Conformément à sa mission principale, le DAPNIA/SEDI développe, pour les expériences de physique, des systèmes électroniques complets (matériel + logiciel) d'acquisition et de traitement de données en temps réel. Or, la taille et la complexité croissantes des expériences de physique conduisent à ce que ces systèmes soient de plus en plus répartis et hétérogènes. Le développement de ces systèmes rencontre donc des problèmes récurrents de conception et de déploiement pour lesquels, aujourd'hui, il n'y a pas au SEDI de méthode suffisamment formalisée d'analyse / conception / mise en œuvre. Plus précisément, l'accroissement du niveau de complexité des systèmes exige l'emploi d'une méthodologie de développement plus explicite (et donc plus formalisée) permettant une conception plus « industrielle ». L'expérience et la recherche en matière de génie logiciel montrent

que l'explicitation d'une telle méthodologie est nécessaire si l'on veut maintenir le niveau de qualité des systèmes produits malgré leur complexité croissante. En d'autres termes, il nous faut des outils conceptuels qui nous permettent d'orienter notre activité de conception / développement vers plus de modularité afin que, d'une part, les composants matériels et logiciels produits soient plus robustes, plus intelligibles, plus maintenables, et d'autre part, que leur processus de développement soit plus parallélisable.

L'industrie du génie logiciel propose déjà des normes, des standards et des outils permettant de rationaliser les processus de développement logiciel. Il semble ainsi acquis que les technologies dites « objet » soient incontournables, en particulier celles normalisées par l'OMG (Object Management Group) autour de la notation UML (Unified Modeling Language), de CORBA (Common Object Request Broker Architecture) et de MDA (Model Driven Architecture) pour ne citer que les domaines présentant un intérêt certain pour le SEDI. En se basant sur ces normes présentes ou à venir, des industriels du développement logiciel proposent déjà des outils relativement aboutis permettant l'analyse et la modélisation des systèmes et, dans une certaine mesure, la génération automatique de code. Certains de ces outils sont plus particulièrement dévolus au domaine du temps réel et sont donc plus susceptibles de présenter un intérêt pour le SEDI.

Cependant, ces outils mettent en œuvre des concepts très généraux pouvant s'adapter à de nombreux domaines d'activité. Aussi, une utilisation réellement efficace de tels outils dans le cadre de l'activité du SEDI ne pourrait-elle faire l'économie d'une réflexion approfondie permettant de spécialiser l'outil en l'adaptant aux problématiques spécifiques à nos processus de développement. De plus, il s'agit d'éviter de mener une réflexion centrée sur des outils sous peine de créer une dépendance conceptuelle et matérielle trop forte envers un vendeur particulier. Nous devons avoir comme souci constant de ne dépendre que des normes et des standards bien établis et d'avoir au maximum la maîtrise de nos outils logiciels. C'est pourquoi, il nous semble qu'il importe avant tout de nous fabriquer une méthodologie de développement indépendamment de tout outil particulier et en utilisant le cadre conceptuel standard proposé par la norme UML, ce qui n'empêche pas que l'on mette en œuvre cette méthodologie à l'aide d'un (ou même plusieurs) outil(s).

MORDICUS est donc un projet de R&D destiné à lancer les bases d'une méthodologie orientée objet fondée sur UML de développement de systèmes d'acquisition / traitement pour le SEDI. Plus précisément, son ambition est de rationaliser le développement de systèmes distribués mixtes matériels et logiciels. En effet, dans les habitudes de travail du SEDI, il n'existe pas aujourd'hui de séparation suffisamment explicite entre la conception de l'architecture distribuée d'un système et celle des fonctionnalités qu'on en attend. En d'autres termes, les ingénieurs conçoivent en même temps la fonctionnalité du système demandé et son implémentation sur un déploiement particulier. Par exemple, la décision d'implanter telle partie du système sous forme matérielle ou logicielle ou de mettre en œuvre le système sur telle topologie de réseau n'est pas clairement fondée sur une réelle distinction entre ce qui relève de la fonctionnalité du système et ce qui n'en est qu'une modalité d'implémentation. Ce flou conceptuel nuit à la qualité des produits fournis notamment en termes de robustesse, d'intelligibilité, de maintenabilité et de portabilité. Il entraîne, par exemple, un mélange entre code fonctionnel et code de communication qui rend les système moins portables et donc moins évolutifs et moins optimisables. Par son manque de souplesse, il entraîne également des surdimensionnements évitables. Il est en effet plus facile d'optimiser un ensemble de petits modules qu'un programme monobloc.

Il s'agit donc de définir un cadre conceptuel pour le développement de systèmes qui clarifie les aspects concernant les spécifications fonctionnelles et ceux relatifs aux spécifications de déploiement / implémentation. Une telle séparation permettrait, par exemple, de porter une application sur une nouvelle plate-forme en se contentant d'adapter l'architecture et le code de déploiement sans toucher au code purement fonctionnel.

Enfin, l'expérience et la recherche dans les technologies objet montrent que le développement de systèmes complexes impose une méthodologie incrémentale : en commençant par l'analyse de l'architecture générale du système à construire, le cycle conception / implémentation / validation est répété de nombreuses fois pour aboutir à l'implémentation de l'architecture détaillée du système. MORDICUS se propose également de produire une méthodologie incrémentale qui commencerait par la spécification purement fonctionnelle du système, puis poursuivrait par de multiples redéploiements successifs sur des architectures de plus en plus complexes pour aboutir finalement au système réel complet. Le but principal de la

méthodologie serait donc d'organiser une séparation suffisante entre aspects fonctionnels et aspects de déploiement afin que les cycles de redéploiements et les cycles d'évolution des fonctionnalités puissent s'effectuer avec un impact limité et contrôlé des uns sur les autres.

VI.E.3 But du projet

1. Développer et formaliser une méthodologie spécifique de développement de systèmes d'acquisition / traitement qui réponde à la problématique ci-dessus.
2. Mettre en œuvre cette méthodologie sous la forme d'un outil logiciel destiné à assister la conception de systèmes d'acquisition / traitement et à permettre la génération semi-automatique de code.
3. Banaliser l'usage des technologies et méthodes objet au sein des équipes du DAPNIA/SEDI. Cet objectif pourra être atteint par la participation –même épisodique– de tous à MORDICUS, en utilisant la méthodologie dans des projets concrets et en exprimant en retour critiques et propositions d'amélioration.

VI.E.4 Description du projet

Le projet s'attachera à expliciter, formaliser et rationaliser les méthodes implicites utilisées au SEDI pour le développement de systèmes. Cette formalisation sera basée sur le modèle objet et utilisera la notation UML en raison de son caractère de standard industriel de fait.

La méthodologie développée s'appuiera sur les motifs récurrents spécifiques à notre métier (systèmes d'acquisition/traitement pour la physique) et l'ensemble de ces motifs sera évolutif selon l'apparition de nouveaux besoins.

L'AGL utilisé pour la mise en œuvre concrète de la méthodologie devra comporter un langage de méta-programmation permettant l'implémentation opérationnelle des motifs. L'AGL Objecteering en est un exemple.

Le code résultant sera exprimé dans les langages java et/ou C++.

Il est impératif que la méthode soit adaptée au développement de systèmes répartis et hétérogènes comprenant notamment des modules appartenant au monde du temps réel et de l'embarqué.

Des documentations décriront la méthode et l'utilisation des motifs dans l'AGL

ainsi que la conception et la mise en œuvre des motifs sous forme de méta-programmes.

VI.E.5 Production

Le projet MORDICUS doit aboutir à un résultat composé de 3 parties distinctes :

1. MORDICUS_FRAMEWORK

Partie logicielle, composée des motifs spécifiques. Ces motifs seront mis en œuvre dans l'AGL commercial pour l'adapter à notre métier.

2. MORDICUS_USER_GUIDE

Manuel utilisateur, qui contiendra la description de la méthodologie de conception / développement, la notation utilisée, et l'usage des motifs dans l'AGL. Ce manuel doit permettre la création d'un code objet correspondant à l'analyse de l'utilisateur.

3. MORDICUS_EXPERT_GUIDE

Un manuel expert décrivant les fondements de la méthodologie et l'architecture interne de l'outil logiciel. Ce manuel sera destiné à ceux qui seront en charge de la maintenance et de l'évolution du produit.

VI.E.6 Formations et publications

Les intervenants principaux du projet devront également assurer la transmission interne des savoirs acquis en assurant des formations liées aux technologies employées (UML, méthodologies et langages orientés objets, méta-programmation). Il s'agit, comme anoncé en début de document, de banaliser et de diffuser ces technologies à travers le service voire le département.

Les résultats du projet qui seront de nature à intéresser la recherche feront l'objet de publications dans les revues spécialisées et de présentations à des conférences.

VI.F PUBLICATION DANS REVUE

Les principes directeurs de cette thèse ont fait l'objet d'une publication dans une revue. Le texte intégral de cet article est reproduit ci-après dans sa version originale. La référence exacte de l'article est:

S. Anvar, F. Terrier, *A UML-based design process for distributed data acquisition and triggering systems in high energy physics experiments*, in IEEE Trans. on Nucl. Sc., part I, pp. 586-594, num. 3, vol. 48, June 2001.

Le texte intégral de cet article est reproduit ci-après dans sa version originale.

A UML-based Design Process for Distributed Data Acquisition and Triggering Systems in High Energy Physics Experiments

Shebli Anvar, François Terrier

Abstract--**We present the first results in the elaboration of a design and development process that tackles recurrent problems encountered when developing acquisition and triggering systems for high-energy physics experiments. These problems include software/hardware frontier definition and the impact of both intrinsic and performance-related parallelism on software development. Based on the UML (Unified Modeling Language) and its extension mechanisms, the process aims at capturing rules, constraints and simple mechanisms that 1) separate functional concerns from deployment specifications, including hardware-software separation, and 2) transparently and automatically derive distribution patterns from system-level definitions. The process is intended to grow progressively into a design framework that will both enforce its rules and constraints and implement design patterns that result from our experience of HEP (High Energy Physics) TDAQ (Trigger and Data Acquisition) system development.**

I. Introduction

THE digital systems developed for data acquisition and real-time processing in HEP experiments keep getting bigger and globally more complex [1], [2], [3]. The constraints on such systems are more stringent than ever both in terms of performance and robustness. To be able to cope with the design and development challenges that such growing complexity and size entail, the engineers of the HEP community have to fathom new methods and tools that make these future systems feasible, cost-effective and maintainable over many years.

Computer science experience and applied research [4]-[9] show that major solutions to system design rationalization always involve the abstraction of processes and patterns specific to the application domain and their implementation as reusable components inside a domain-specific framework. We have therefore started a reflection leading us to elaborate on a design methodology specifically adapted to data acquisition and trigger systems such as those developed in HEP-like experiments. We have begun expressing the results of our reflection in the form of a development process or 'conceptual' framework that will be the basis for the progressive development of an 'actual' design framework that will include generic design architectures and reusable components [7]-[9]. We believe that a successful design framework cannot be built *a priori* with components that are supposed to be generic from the beginning. Instead, we try, more pragmatically, to capture rules and constraints that will seemingly enhance design quality, using the experience of HEP TDAQ designers and in the light of new concepts, methodologies and technologies provided by the industrial and research communities of embedded and real-time electronic systems. Hence, the purpose of this paper, which is to discuss a number of concepts and ideas related to recurrent problems encountered in the design of TDAQ systems.

In section II we discuss the necessity of a separation of concerns in the overall development process, between functional and distribution issues. This necessity pertains to the need for multiple distribution schemes, flexibility in the definition of hardware/software frontier and difference in design impacts of performance-related parallelism and intrinsic parallelism. Distribution issues are raised by the necessity to implement the functional specifications of a system over computing resources that are somehow "spread-out." What we call "deployment" is any scheme that aims at realizing this implementation.

In section III, we briefly justify our choice of the UML notation and, using a toy example, we present two classes of UML diagrams respectively devoted to functional and deployment specifications. We also point out some UML extensions needed for our purposes.

In section IV, we present a scheme for the automation of distribution-specific patterns through the analysis of functional and deployment specifications.

Section V is devoted to a discussion on the choice of development and distribution technologies for a TDAQ framework in the context of HEP experiments, especially taking into account maintainability and evolution in projects with lifetimes that span over many years.

We conclude the paper in section VI by showing the benefits of these first ideas for a TDAQ conceptual design framework and presenting future research plans on the subject.

Manuscript received November 3, 2000.

S. Anvar is with the Département d'Astrophysique, physique des Particules, physique Nucléaire et Instrumentation Associée, CEA Saclay, F-91191 Gif-sur-Yvette, France (telephone: +33(0)169087832, e-mail: Shebli.Anvar@cea.fr).

F. Terrier is with the Département des Technologies des Systèmes Intelligents, LIST, CEA Saclay, F-91191 Gif-sur-Yvette, France (telephone: +33(0)169086259, e-mail: Francois.Terrier@cea.fr).

II. Separating Functional Design and System Deployment

A. Multiple Distribution Schemes During Design and Development

HEP TDAQ systems are large-scale, complex systems that necessitate a progressive and incremental design process. Basically, we have physics detectors whose topology and qualities are essentially determined by the scientific principles on which the experiment is based. They produce signals that are digitized by the front-end electronics. The function of the TDAQ system is to process the data produced by the front-end electronics and store the results for further scientific analysis. Therefore, the TDAQ subsystem designers are basically given the geometrical organization of an input data-flow together with a quantitative estimation of its extent. From that, they must design a subsystem that carries out the required processing and/or acquisition.

The development process goes through a general specification phase in which the principle of the final distribution scheme is devised by specifying the processing nodes (participants) and their interconnections (topology). The actual design and development of the subsystem then consists in a considerable number of *successive* hardware/software developments, design refinements, tests and optimizations. We might say that the final system will progressively grow from a seminal simple system through successive "complexifications." This is an *iterative* process in which each cycle will produce an *intermediary distribution scheme*. For instance, we might begin with a purely functional program running on a single computer and then deploy the program on a small multi-node network, then on a heterogeneous network involving more specific processor types needed in different parts of the system, etc. until the system reaches its final size and complexity. Each cycle might involve redeployments, hardware/software design and development, software porting, and actual tests and measurements of the corresponding setup that lead to further optimizations. As a consequence, before the final distribution scheme is actually implemented, the TDAQ designers have to implement numerous smaller, tentative schemes. During this iterative process, the functional design of the system is likely to evolve at a different (slower) pace than the distribution architecture. Consequently, in order to minimize re-designs and code re-development, a design process for TDAQ systems should provide for means of *separating* these two aspects, so that evolutions of the distribution scheme entail minimal modifications in the system's functional architecture.

B. The Hardware/Software Frontier Problem

High-energy physics experiments always depend on custom-made front-end detectors that need specific electronic systems for read-out and acquisition. Behind the physics detectors – such as photomultipliers, wire chambers or CCDs – specific hardware is necessary to give form to detector signals, digitize and do some real-time processing on the resulting data flow. However, specific hardware must stop *as soon as possible* in the acquisition chain in order to pass on the data to flexible (i.e. software) subsystems running over COTS (Commodity Off-The-Shelf) electronics and a standard OS (Operating System). The necessity of such flexibility stems from the fact that time scales of both the design and the exploitation of HEP physics experiments tend to extend over many years, which calls for possibilities of modifying designs even during the development cycle in order to keep up with technological advances and evolving standards. Consequently, the *frontier between hardware and software* components inside a subsystem *should be decided for as late as possible* in the design process. This calls for techniques that allow the *design* process to go on without having to specify which components will be hardware and which will be software. The results of this design process must be then *deployed* over a specific hardware configuration. The core of any framework solution tackling this problem will therefore be a *separation of concern* between functional system design and deployment design. In other terms, the hardware/software frontier determination can actually be treated as a *deployment* problem: if an object is deployed as a hardware component, then no code is generated for it (or firmware code such as VHDL code), and the designer must only define a precise interface for communication between code and hardware.

C. Intrinsic and Performance Distribution

The data acquisition systems and their associated trigger systems are usually *distributed by nature*, because detectors are spread out — sometimes over great distances. In collider experiments, for instance, tens of thousands (if not millions) of electronic channels must be digitized, data-formatted, processed and stored [3]; in some astroparticle experiments such as ANTARES [10], detector nodes are spread over volumes of millions of cubic meters. We call this kind of distributed feature "intrinsic distribution," i.e. distribution that arises because of "intrinsic parallelism."

In addition to that, the data flows in HEP TDAQ systems are often considerable and consequently call for processing power that cannot be provided by single machines. In such systems, intrinsic distribution is therefore accompanied by "performance distribution," that is, distribution that arises due to the scaling up of processing power through parallelization, such as in computing farms. In short, whether intrinsically or for the sake of processing power, HEP TDAQ systems are most often massively distributed systems, and the distinction between performance distribution and intrinsic distribution has non-negligible impact on their design.

1) Design Impacts of Intrinsic Distribution

Intrinsic distribution stems from the spreading out of processing nodes that are close to the front-end electronics. These nodes contain specific electronics and are often embedded and hard –if not impossible– to access. As a consequence, they may evolve frequently during the design process but are not liable to evolve much after that. Intrinsic distribution in HEP TDAQ systems is therefore quite static throughout the detector lifetime. Moreover, the computing nodes associated to intrinsic distribution are

often of embedded nature because of their proximity to the detector. As a consequence, they are very difficult (if not impossible) to reach physically. Failures in these nodes will tend to be long-term failures if not permanent ones. That is why the intrinsically distributed part of a TDAQ system often calls for more robust designs.

2) Design Impacts of Performance Distribution

Performance distribution, on the other hand, is mostly found in the form of processor farms that are accessible and easily upgradeable. Performance-distributed subsystems are therefore likely to have a fast evolution rate in order to keep up with technological advances. Also, their scalability makes them less sensitive to failures, as processor failures do result in a degradation of performance but are less liable to cause a complete breakdown. Of course, sensitive devices such as key servers or switches can cause global breakdowns if they fail; however, they are usually easily replaceable because they are within easy reach of human operators.

3) Impact on Design Process

These qualitative differences in design requirements between intrinsically distributed and performance distributed subsystems imply that any design framework for HEP TDAQ systems must allow for such a distinction. Indeed, the design process must be able to apply specific procedures for each.

III. Functional and Deployment Diagrams

A. Why Use the UML?

An HEP TDAQ design framework that enforces the separation of concerns discussed in subsections A and B should clearly distinguish two classes of specifications, one for functional and one for deployment concerns. The UML notation [11] provides for diagrams that are quite adapted for that purpose. It also provides for standard extension mechanisms for specializing the notation and adapting it the specific domains (such as HEP TDAQs in our case). Moreover, the UML is today universally recognized as the definitive standard for object systems modeling [12], [13] and all modern methods and software development frameworks are UML-based (e.g. [14]). We have therefore decided to base our own TDAQ design process on UML notation and modeling. This will ensure maximum compatibility with COTS development frameworks and prevent us from heavily relying on proprietary languages and notations.

B. The Two Classes of UML Diagrams

The twofold system specification in our TDAQ process would rely on two classes of UML diagrams: the first class would be devoted to functional design and would include all the UML static and dynamic specification diagrams such as class, collaboration, sequence, activity or state transition diagrams. The second class of diagrams would be essentially based on the UML deployment and/or component diagrams. Ideally, the system designer should be able to: 1) define the system as *one* program in the form of a set of interacting functional objects that implement the processing algorithms that the system is expected to perform and 2) specify *many* deployment schemes that represent as many ways of running the program on different network topologies.

C. Diagrams for a Simple Example

Let us consider, for instance, a Simplistic Astrophysical Multi-Spectral Analysis (SAMSA) system that processes images coming from two detectors attached to a telescope. Detector 1 is sensitive to infrared light and detector 2 to visible light. The system must first bundle pairs of images, then find correlation patterns between the infrared and visible images and then store the images and the results in a compressed format.

1) Functional Diagrams

Fig. 1 and Fig. 2 respectively show the sequence diagram that represents the typical call sequence between objects of the system and the class diagram that defines the static architecture of the system. Each detector is coupled with a DetectorReadout object that produces digitized data (images) and sends them (through an asynchronous *acquire(RawImage)* call) to a PairBuilder object. PairBuilder merges each pair of images into one Pair data object and then sends the result to a PairProcessor object through a *correlate(Pair)* call for correlation computation. The Pair object together with the computation result Correlation are then sent to Storage through a *store()* call. These two diagrams are clearly functional specification diagrams, as they define the objects we need and how they interact but do not specify on what hardware infrastructure they are deployed. We could go further and explicitly write all the code attached to the specified classes (for instance, the full code of the PairProcessor.correlate() method). Then, for that single set of functional specifications, the designer imagines two successive different deployment schemes expressed in Fig. 3 and Fig. 4.

2) Deployment Diagrams

Fig. 3 features a deployment where data are produced by "readout," a software component simulating the telescope readout device; the data are then sent through a network connection to the "processor object" component in which objects PairBuilder, PairProcessor and Storage are implemented. A number of implementation details are stated in this diagram, namely that 1) the PairBuilder, PairProcessor and Storage objects are implemented by the same component on a the same PC-Linux node and consequently run in the same address space whereas the DetectorReadout object runs in another one on another machine; 2) the two nodes running the "processor" and "readout" components are linked by a bi-directional communication package called "myCORBA." The framework can therefore automatically deduce that all method calls between PairBuilder, PairProcessor and Storage objects take place as classical function calls (i.e. through normal post-compilation link), whereas method calls between DetectorReadout and other objects have to go through proxy objects as defined by the "myCORBA" package.

Fig. 4 represents a more realistic deployment that is closer to the final system. As expected on most TDAQ systems, the readout of the detector is carried out by a

specific firmware, here implemented on an FPGA. The data merging takes place in the "builder" component on an embedded processor running a RTOS (Real-Time Operating System), whereas the correlation computing and the storage are carried out on a PC farm running Linux. Here, classical method calls are only between PairProcessor and Storage objects, the other ones having to go either through a CORBA package or a protocol based on interrupts and shared memory. Moreover, PairProcessor and Storage objects are distributed over a PC farm, which calls for a processor farm management system.

3) Needed UML Extensions

Our specific interpretation of node associations as pointers to communication packages is a first implicit UML extension. Other than considering the association name as a communication package name, we also agree upon interpreting the navigability of node associations (arrows at one or both ends of the association line) as directionalities that the communication package can support. In Fig. 4, for instance, the communication link between the ALTERA20K node and the PowerPC node is mono-directional. In other words, only a sender in the "readout" component and a receiver in the "builder" component have to be implemented, as opposed to the CORBA package that supports communications in both directions between nodes PowerPC and PC-Linux. We could render our interpretation of node associations more explicit by creating a new stereotype for them (such as **<< comm >>**) in order to avoid any confusion with other interpretations, but we must also try to limit UML extensions to the most needed features and refrain from terminological inflation.

Apart from that specific interpretation of node associations, we have introduced a few specialized stereotypes to be able to specify unambiguously some features in our deployment specifications.

The **<< impl >>** stereotype over "use" dependency links is an extension of the UML that we need to specify object implementations in the form of components running on a specific node.

The purpose of the **<< fpga >>** stereotype over a node name is to introduce the notion of firmware in system deployment; this allows us to specify deployments featuring objects implemented in hardware (or rather firmware). Any code generator included in the framework would then know which source code type is related to which objects. For instance, in Fig. 4, the "readout" component runs on an FPGA node: a code generator would then use the associated VHDL files to generate the code attached to the DetectorReadout class. In a more distant future, once the UML action language [15] is sufficiently specified by the OMG and developed by the software industry, it would be natural for any code generator to be able to translate action language statements into the right source language (VHDL, C++, Java, etc.) for each object, according to its placement in the deployment diagram. For the moment, VHDL programs have to be hand-written, but we can already follow the rules concerning separation of concerns so that generic framework components can be factored out from these programs.

The **<< farm >>** stereotype over a node is of a more subtle nature: it means that although an object such as PairProcessor is seen as *one* object in the system (see in Fig. 2 the '1' cardinality in the association between classes PairBuilder and PairProcessor), it is implemented in Fig. 4 on *many* nodes in parallel for the sake of performance and/or failure tolerance.

IV. Automatic Control of Distribution Patterns

Although the specification of design architecture using a formal language such as UML diagrams is in itself useful for productivity and software quality [13], [14] our goal is to be able to achieve more than that. Indeed, apart from presenting a constraining environment to enforce rigor in design, the TDAQ design framework should also, through the cross-analysis of the specification diagrams, 1) check the consistency of our design according to domain-specific criteria and 2) automatically execute model transformations that correspond to the application of recurrent patterns.

A. Consistency Check

Numerous consistency checks can be run on a UML design tool, and most of the industry's CASE tools such as Rational Rose [12] or Objecteering [13] include a number of them, such as namespace and scope coherence. We might include some more that are directly attached to our development model. For instance, the analysis of functional diagrams points to objects that need to communicate with each other (such as PairBuilder and PairProcessor in Fig. 1 and Fig. 2). Therefore, on deployment diagrams, we can check if any two nodes that run implementations of two such objects are indeed associated. Other consistency checks are possible, especially during automatic model transformation, and they include checking the existence and conformity of communication packages.

B. Automatic Model Transformation

When a remote communication between two objects is detected, the software organization must be modified accordingly. Let us consider again the SAMSA example in its Fig. 4 deployment. From Fig. 4, the tool can readily deduce the existence of two address spaces, one on an RTOS-running PowerPC, and the other on a Linux PC. It can therefore create one directory associated to each one of them, corresponding to one executable binary for each.

1) Proxy Generation

Let us focus on PairBuilder and PairProcessor objects. In the PowerPC-RTOS directory, the PairBuilder code is generated using directly the code developed in the functional specification. The same is done in the PC-Linux directory for the PairProcessor code. However, special code must be inserted in the PowerPC-RTOS directory so that all PairBuilder calls to PairProcessor are transparently compiled and run without modification of the PairBuilder code. As expressed in Fig. 5, that special code consists in a "proxy" PairProcessor, in the sense defined by ORB architectures (such as CORBA) [6]: it is a class that has the same name as the original class (PairProcessor), the same interface, but not the same implementation code, as the implementation of all methods consists only in the

marshalling of parameters, their sending to the real object through the communication package, waiting for the real execution to complete, and finally returning the return value to the calling object.

2) Skeleton Generation

Symmetrically, as expressed in Fig. 6, a "skeleton" code (in the CORBA sense, see [6]) is added to the PC-Linux package, i.e. an object that listens to the communication link for execution requests and translates them into actual method calls on PairProcessor. It should be noted that the communication package does not only provide the framework with precise proxy/skeleton production rules, it also has to implement an initialization procedure that correctly instantiates them (a more thorough examination of the instantiation question is beyond the scope of this paper).

3) Farm Manager Generation

The functional diagrams state that the PairProcessor object is supposed to be logically one single object. However, the **<< farm >>** stereotype in Fig. 4 indicates that it is implemented as many identical components running on parallel nodes. Consequently, before being able to send the "correlate" request, a PairBuilder object must first determine which component will be the receiver. Inserting a new management object between PairBuilder and the PairProcessor proxy naturally solves the problem (Fig. 7). In other words, the PairBuilder object will see the farm management object as a genuine PairProcessor (same interface) but its *correlate()* requests will be routed to the right proxy according to the farm management policy implemented inside the manager object. In other words, the model transformation fools the caller (PairBuilder object) by presenting it the interface it expects and hiding the parallelism management issue from it, thus preserving the separation between functional code and farm management code.

V. Which Software Technology to Use?

The general cost-reduction pressure that is always present in HEP projects calls for the use of COTS products and industry standards. Indeed COTS theoretically allows for less development and maintenance effort since these activities can in principle be partially delegated through the use of industrial ready-made products. But the same concerns arise when COTS becomes synonym of dependency towards a specific vendor. Indeed, development and maintenances efforts are liable to rise abruptly if the vendor we depend on ceases to support a product or simply ceases to exist. Therefore, we consider that advantages of COTS should be evaluated against dependency problems.

In any case (COTS or not), dependency concerns are less constraining if software sources are accessible (and understandable). The necessity of having access to software sources also stems from the need for performance optimizations, and porting constraints. If the software architecture is well designed and modular enough, performance bottlenecks can be diagnosed relatively easily and more effort can be put in the optimization of small modules. At the same time, good architectural design allows for easy porting of the software over evolving platforms. A clear example of successful software architecture both in terms of porting and optimization capabilities is the open-source CORBA middleware "TAO" developed by the University of Washington [6], [19], [20]. We intend to use this middleware in our own framework because 1) TAO's open-source model together with an abundant documentation allow us to avoid re-developing important amounts of software without being dependent on the goodwill of a vendor, and 2) TAO implements a successful industry standard (i.e. CORBA) that does not depend on any proprietary choice. Moreover, the design of TAO has been strongly constrained to support real-time distributed systems, as opposed to most COTS ORBs, which are known to behave poorly in real-time environments [6], [14].

As for the framework implementation of the design rules and constraints that constitute the core of our process (such as systematic transformations), we need a pattern language based on the UML. As stated in section IV, real productivity gains can be obtained if the framework is implemented on a CASE tool with automatic code-generation capabilities. "Objecteering", by Softeam, is the tool we have chosen for this implementation and is the only vendor-dependant product that we intend to use. The main feature that has caught our attention is a "java-like" pattern language (the J language) that allows the users to freely implement their own design patterns by working directly at the metamodel level, i.e. *before* the code generation level (of course code-generation is, *a fortiori*, also modifiable). In other terms, the tool supports the UML extension mechanisms, and the automatic model transformations that translate TDAQ design patterns can be readily programmed at the metamodel level. In addition, the adaptability of this tool to the modeling of real-time systems has been thoroughly validated by the development of the ACCORD framework for real time system prototyping in another laboratory of our institution (CEA-LIST, Software for Process Safety Laboratory) [16], [17]. As witnessed by the fact that "Rational Rose" [12] (the most popular tool in the industry) does not support metamodel programming, CASE tools with metamodel programming capabilities are quite rare. Apart from Objecteering, a research project called UMLAUT [18], developed by INRIA/IRISA at Rennes in France features analogous capabilities although not at the same industrial maturity yet. Since it is an open source research project, switching to UMLAUT is certainly an option that is worth studying. Since both tools support "XMI" the XML extension devoted to the exchange of UML models, the switch should not be too painful.

Eventually, we intend to map the core concepts of our TDAQ design process to an official UML profile, that is, a domain-specific extension of the UML adapted to the design and development of HEP-like TDAQ systems.

VI. Conclusion and Future Research

A. Simultaneous Implementations

A TDAQ design process that follows the ideas presented in this paper would above all constitute a conceptual

environment enforcing an iterative design and development philosophy. Apart from guiding the designer along a progressive path from simple seminal setups to the fully integrated real TDAQ system, it allows the simultaneous maintenance of different implementations of the system. The best example of such a feature stems from the need for physics analysis to maintain a functional simulation of the TDAQ system. Indeed HEP experiments almost always need to determine the precise effect of the TDAQ system on data quality (especially for triggers, but this remark holds for any computation that filters or modifies physics data) by cross-examining the functional behavior of the system with Monte Carlo simulations of the detector. By itself, such a need calls for the coexistence of at least two deployments of the same system: the real system and a purely functional deployment on a single machine. It may also be necessary to maintain intermediary deployments corresponding to a subset of the TDAQ system for post-production debugging and maintenance.

B. More Automated Model Transformations

Model transformations other than class diagram modifications will also be necessary, especially the modification and/or creation of state diagrams included in special objects such as skeletons or farm managers. Indeed such recurrent problems are solved by patterns that are not always restricted to the static class structure of the system and often involve dynamic specifications too, and we intend to go further in this direction. Also, the notion of a distributed object (like the PairProcessor object) over a farm of processing nodes deserves to be extended to other useful concepts such as the distributed state-machine over all the nodes of the system.

C. Performance Analysis

It is worth pointing out that the Object Management Group (OMG) –the official institution in charge of the UML standard– is presently conducting work on the possibility of performance analysis on UML models. This work has been initiated under the "Scheduling, Performance and Time" UML profile proposal [21], [22]. The idea is to give the possibility to designers to include all the information that is required for a performance analysis inside the UML application model (in the form of implementation hints). This approach is fully compatible with the modeling process that we propose in the paper and should therefore leave the door open for our framework to include performance analysis modules based on future standard UML profiles.

VII. REFERENCES

[1] M. Jacob, "From basic research, its primordial goal, to technological transfers and industrial spin-offs," in *Proc IEEE 10th Real Time Conference*, 1997, pp. xii-xvii.

[2] J. Knobloch, "ATLAS computing," in *Proc. CHEP97*, 1997.

[3] The ATLAS Collaboration.. (1996, Dec.). *ATLAS computing technical proposal*, (*CERN/LHCC* 96-43) [Online]. Available: http://atlasinfo.cern.ch/Atlas/GROUPS/SOFTWARE/TDR/html/

[4] J.-P. Briot, R. Guerraoui, "Objets pour la programmation parallèle et répartie: intérêt, évolutions et tendances," *Technique et science informatique*, vol. 15 – n°6, pp.765-800, 1996.

[5] M. D. Lubars, N. Iscoe, "Frameworks versus libraries: a dichotomy of reuse strategies," in *Proc. WISR'93 6th Annual Workshop on Software Reuse*, 1993.

[6] D. C. Schmidt, D. L. Levine, C. Cleeland, "Architectures and patterns for developing high-performance, real-time ORB endsystems," in *Advances in Computers*, vol. 48, M. Zelkowitz, Ed. Academic Press, July 1999.

[7] H. A. Schmid, "Systematic framework design," *Communications of the ACM*, Vol. 40 n° 10, Oct. 1997.

[8] J. Bosch, "Specifying frameworks and design patterns as architectural fragments," in *Proc. TOOLS ASIA '98*, 1998.

[9] D. Roberts, R. Johnson, "Evolving frameworks – a pattern language for developing OO frameworks," in *Pattern Languages of Program Design 3*, R. Martin, Ed. Addison-Wesley, 1998.

[10] S. Anvar, H. Le Provost, F. Louis, "The ANTARES offshore data acquisition: a highly distributed, embedded and COTS-based system," presented at the *2000 NSS/MIC Symposium*, Lyon, France, 2000.

[11] The Object Management Group. (2000, Mar.). OMG Unified Modeling Language Specification. (version 1.3) [Online]. Available: http://www.omg.org/technology/documents/formal/unified_modeling_language.htm.

[12] T. Quatrani, "Visual modeling with Rational Rose 2000 and UML," Addison-Wesley, 1999.

[13] Philippe Desfray. UML profiles and the J language, total control over application development using UML. Softeam. Paris, France. [Online], Available: http://www.softeam.fr/us/smot_uml_white.htm.

[14] F. Kuhns, D. C. Schmidt, D. L. Levine, "The performance of a real-time I/O subsystem for QoS-enabled ORB middleware," in *Proc. DOA'99 Distributed Objects and Applications*, 1999.

[15] The Object Management Group. (1999, Sept.). Action Semantics for the UML – Request for Proposal. [Online]. Available: http://www.projtech.com/pubs/xuml/rfp.pdf

[16] A. Lanusse, S. Gérard, F. Terrier, "Real-time modeling with UML: the ACCORD approach," *Lecture Notes in Computer Science*, vol. 1618 pp. 319-335, 1999.

[17] S. Gérard, "Modélisation UML exécutable pour les systèmes embarqués de l'automobile," University of Evry, Evry, France, PhD Rep., Oct. 2000.

[18] M. Ho, J-M. Jézéquel, A. Le Guennec, F. Pennaneac'h , "UMLAUT: an extensible UML transformation framework," in *Proc. 14th IEEE International Conference on Automated Software Engineering*, 1999.

[19] D. C. Schmidt. The Adaptative Communication Environment (ACE). [Online]. Available: http://www.cs.wustl.edu/~schmidt/ACE.html

[20] D. C. Schmidt. The ACE ORB. [Online]. Available: http://www.cs.wustl.edu/~schmidt/TAO.html

[21] The Object Management Group. UML profile for scheduling, performance and time, request for proposal. (1999, Mar. 13). [Online]. Available: ftp://ftp.omg.org/pub/docs/ad/99-03-13.pdf

[22] B. Selic, B. Douglass, A. Moore, M. Bjorkander, M. Gerhardt, B. Watson. Response to the OMG RFP for schedulability, performance, and time. (2000, Aug. 14). [Online]. Available: ftp://ftp.omg.org/pub/docs/ad/00-08-04.pdf

VIII. Figures and Captions

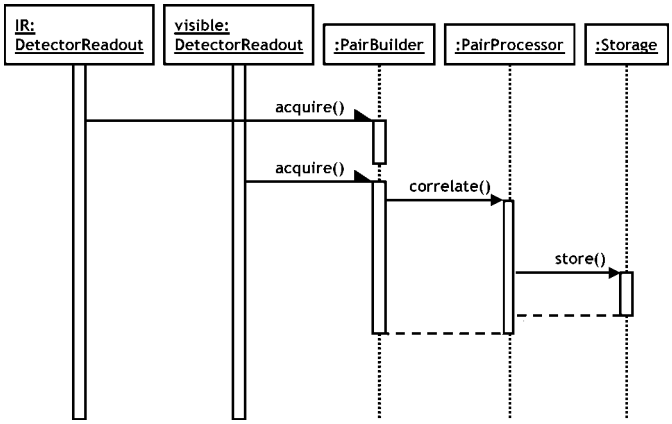


Fig. 1. Sequence Diagram of SAMSA System.

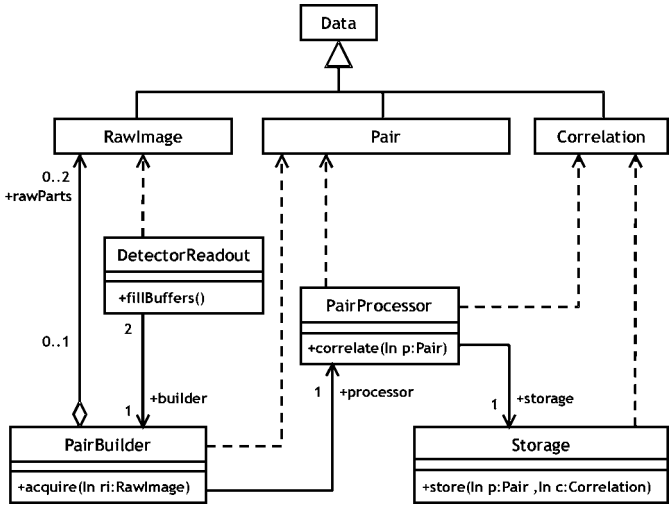


Fig. 2. Class Diagram of SAMSA System.

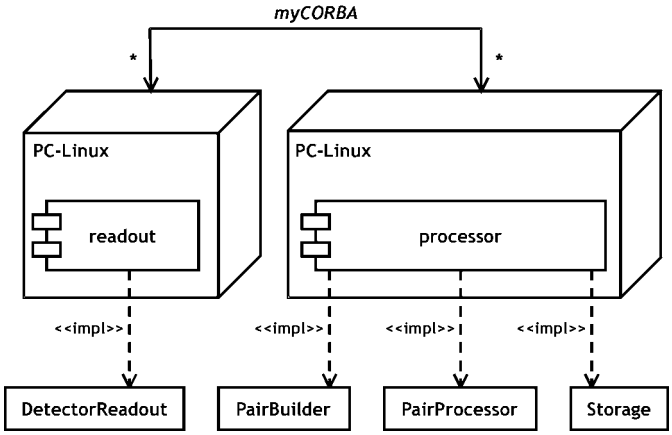


Fig. 3. First Deployment of SAMSA System.

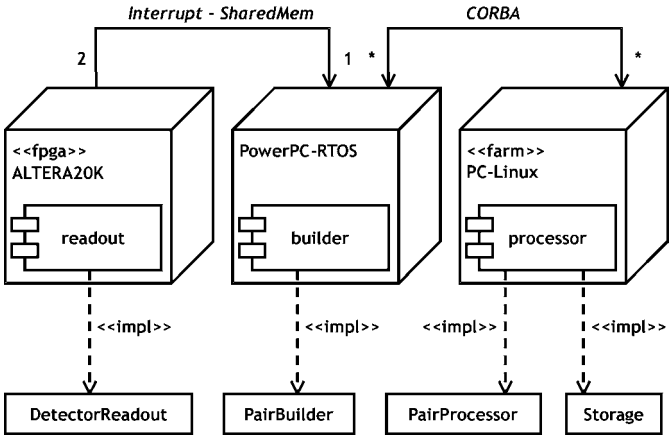
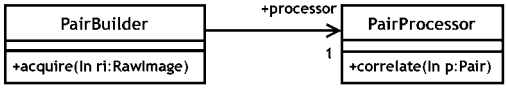


Fig. 4. Second Deployment of SAMSA System.



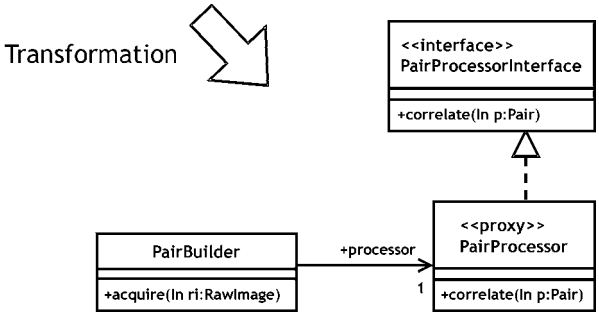


Fig. 5. SAMSA Model Transformation on PowerPC-RTOS Side.

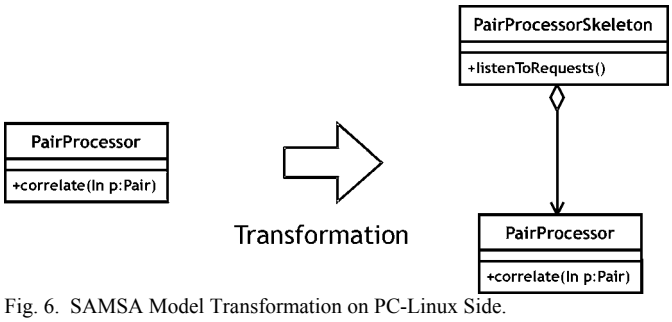


Fig. 6. SAMSA Model Transformation on PC-Linux Side.

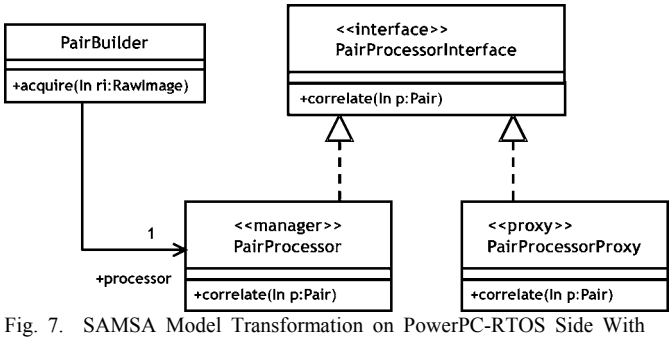


Fig. 7. SAMSA Model Transformation on PowerPC-RTOS Side With Farm Management.

VII

RÉFÉRENCES

Références A: Sites institutionnels

[A-1] CERN, Organisation Européenne pour la Recherche Nucléaire.
http://www.cern.ch/

[A-2] ESO/VLT, European Southern Observatory, the Very Large Telescope Project.
http://www.eso.org/projects/vlt/

[A-3] ANTARES, Astronomy with a Neutrino Telescope and Abyss environmental RESearch.
http://antares.in2p3.fr/

[A-4] DESY, Deutsches Elektronen Synchrotron.
http://www.desy.de/

[A-5] ESA, European Space Agency, Scientific Programme.
http://sci.esa.int/

[A-6] BNL/RHIC, Brookhaven National Laboratory, Relativistic Heavy Ion Collider.
http://www.rhic.bnl.gov/

[A-7] FERMILAB, Fermi National Accelerator Laboratory.
http://www.fnal.gov/

[A-8] KEK, High Energy Accelerator Research Organization.
http://www.kek.jp/

[A-9] SLAC, Stanford Linear Accelerator Center.
http://www.slac.stanford.edu/

[A-10] VIRGO, Interferometer for Gravitational Waves Detection
http://www.virgo.infn.it/

[A-11] AMANDA, Antartic Muon And Neutrino
http://amanda.berkeley.edu/

[A-12] ATLAS, A Toroidal LHC ApparatuS.
http://greybook.cern.ch/programmes/experiments/ATLAS.html

[A-13] CMS, Compact Muon Solenoid.
http://greybook.cern.ch/programmes/experiments/CMS.html

[A-14] ALEPH, Apparatus for LEp PHysics
http://greybook.cern.ch/programmes/experiments/ALEPH.html

[A-15] NA48, CP violation measurement.
http://greybook.cern.ch/programmes/experiments/NA48.html

[A-16] AUGER, The Pierre Auger Observatory.
http://www.auger.org/

[A-17] BABAR, CP Violation in the B0 Meson System.
http://www.slac.stanford.edu/BFROOT/www/Public/index.html

[A-18] CHORUS, CERN Hybrid Oscillation Research apparatUS
http://choruswww.cern.ch/Public/welcome1.html

[A-19] OMG, The Object Management Group.
http://www.omg.org/

Références A: Sites institutionnels

[A-20] ACE, The Adaptative Communication Environment, an OO Network Programming Toolkit.
http://www.cs.wustl.edu/~schmidt/ACE.html

[A-21] Altera Corporation.
http://www.altera.com

[A-22] Xilinx, Incorporated.
http://www.xilinx.com

[A-23] CERN Program Library.
http://wwwinfo.cern.ch/asd/cernlib/

[A-24] GEANT - Detector Description and Simulation Tool.
http://wwwinfo.cern.ch/asd/geant/index.html

[A-25] PAW - Physics Analysis Workstation.
http://paw.web.cern.ch/paw/

[A-26] The GEANT 4 Project.
http://wwwinfo.cern.ch/asd/geant4/geant4.html

[A-27] ROOT, An Object Oriented Data Analysis Framework.
http://root.cern.ch/

[A-28] Objecteering, the UML Case Tool by Softeam,
http://www.objecteering.com/

Références B: Documentation en ligne

[B-1] D. Schlatter, P. Perrodo, O. Schneider and A. Wagner,
ALEPH in Numbers, octobre 1996.
http://alephwww.cern.ch/ALEPHGENERAL/reports/alephnum/alephnum.html/alephnumbers.ps

[B-2] The ANTARES Collaboration,
A deep sea telescope for high energy neutrinos – Proposal for a 0.1 km^2 detector, mai 1999.
http://antares.in2p3.fr/Publications/proposal/proposal99.html

[B-3] P. Sphicas,
Trigger and DAQ at LHC, CERN academic training, février 1998.
http://www.cern.ch/Training/ACAD/sphicas.pdf

[B-4] G.D. Barr, P. Buchholz, R. Carosi, D. Coward *et al.*,
Proposal for a precision measurement of ε'/ε *in CP violating* $K^0 \rightarrow 2\Pi$ *decays*, juillet 1990, CERN/SPSC/90-22 SPSC/P253.
http://na48.web.cern.ch/NA48/Welcome/proposal.pdf

[B-5] S. Loucatos,
Electronics and DAQ for the ANTARES underwater neutrino detector,
présenté à "IEEE NPSS Real Time Conference", juin 1999.
http://antares.in2p3.fr/Publications/conferences/1999/SLoucatos-RT99.pdf

Références B: Documentation en ligne

[B-6] A. Kluge,
The Hardware Track Finder Processor in CMS at CERN, juillet 1997.
http://akluge.home.cern.ch/akluge/work/cms/thesis/thesis.pdf

[B-7] T. Cass,
Computing at CERN – Lecture 3 – Looking forwards,
cours d'été du CERN, 1999.
http://tnt.home.cern.ch/tnt/ssl/1999/Future/Future.pdf

[B-8] S. Schanne,
Mesure du rapport d'embranchement de la désintégration $K_L \rightarrow \mu^+\mu^-\gamma$ et développement d'un système de déclenchement dans l'expérience NA48 au CERN, octobre 1997.
http://www-dapnia.cea.fr/Doc/Publications/Archives/spp-97-1006.pdf

[B-9] S. Anvar, J. Cogan, P. Debu, A. Formica, H. Le Provost, I. Mandjavidze *et al.*
Online track reconstruction and level 2 triggering in NA48,
présenté à "IEEE Nuclear Science Symposium", 1999.
www-dapnia.cea.fr/Doc/Publications/Archives/99-06.pdf

[B-10] S. Loucatos,
ANTARES, towards a km^3 underwater neutrino detector,
présenté au "Dubna Non Accelerator New Physics Workshop", juillet 1997.
http://antares.in2p3.fr/Publications/conferences/1997/
unreg_Loucatos_1997.ps.gz

[B-11] The Object Management Group.
OMG Unified Modeling Language Specification - (version 1.3), mars 2000.
http://www.omg.org/technology/documents/formal/
unified_modeling_language.htm

[B-12] The Object Management Group,
OMG Unified Modeling Language Specification - (version 1.4), septembre 2001.
http://www.omg.org/cgi-bin/doc?formal/01-09-67

[B-13] The Object Management Group,
The Common Object Request Broker Architecture Specification - (version 2.4.2), mars 2000.
http://www.omg.org/cgi-bin/doc?formal/01-02-01.pdf

[B-14] C. Cârloganu,
ANTARES, status and perspectives, présenté à "Conference On Cosmology And Particle Physics", Verbier, juillet 2000.
http://www.nikhef.nl/~o49/papers/CAPP2000.pdf

[B-15] P. Bock, J. Carr, S. De Jong, F. Di Lodovico, E. Gross, P. Igo-Kemenes *et al.*,
Lower bound for the standard model Higgs boson mass : combined results of the four LEP experiments, CERN/LEPC 97-11, LEPC/M 115, Nov 3, 1997.
http://lephiggs.web.cern.ch/LEPHIGGS/papers/TN-518/lepcnote.ps

Références B: Documentation en ligne

[B-16] A. Thomas,
Enterprise Javabeans technology, server component model for the Java platform, décembre 1998.
http://java.sun.com/products/ejb/pdf/white_paper.pdf

[B-17] Sun Microsystems,
JDBC data access API.
http://java.sun.com/products/jdbc/index.html

[B-18] Sun Microsystems,
Java Servlet technology.
http://java.sun.com/products/servlet/index.html

[B-19] IBM,
XML Metadata Interchange.
http://www-4.ibm.com/software/ad/standards/xmi.html

[B-20] P. Desfray,
UML profiles and the J language, total control over application development using UML,
http://www.softeam.fr/us/smot_uml_whit

[B-21] J. Kleinöder, M. Golm,
MetaJava — A platform for adaptable operating-system mechanisms, presenté à ECOOP'97 en workshop, août 1997.
http://www4.informatik.uni-erlangen.de/TR/pdf/TR-I4-97-12.pdf

[B-22] Umar Syyid,
A tutorial introduction to the Adaptative Communication Environment (ACE)
http://www.cs.wustl.edu/~schmidt/PDF/ACE-tutorial.pdf

[B-23] The ATLAS Collaboration,
ATLAS computing technical proposal, CERN/LHCC 96-43, décembre 1996.
http://atlasinfo.cern.ch/Atlas/GROUPS/SOFTWARE/TDR/html/

[B-24] F. Hubaut,
Optimisation et caractérisation des performances d'un télescope sous-marin a neutrinos pour le projet ANTARES, rapport de thèse, CNRS - Centre de Physique des Particules de Marseille - Marseille, avril 1999.
http://antares.in2p3.fr/Publications/thesis/1999/Hubaut-phd.pdf

[B-25] P. Amram, S. Anvar, E. Aslanides, J-J. Aubert, R. Azoulay, S. Basa, *et al.*,
Background light in potential sites for the ANTARES undersea neutrino telescope, in "Astroparticle Physics", vol. 13 (2000), pp. 127 à 136, mai 1999.
http://antares.in2p3.fr/Publications/papers/astro-ph-9910170.ps.gz

[B-26] Texas Instruments Inc.,
TMS320C40 User's Guide, mai 1999.
http://www-s.ti.com/sc/psheets/spru063c/spru063c.pdf

Références B: Documentation en ligne

[B-27] D. Lachartre, F. Feinstein,
Application specific integrated circuits for antares offshore front-end electronics, présenté à "BEAUNE 99" (2nd international conference on new developments in photodetection, Beaune, France), juin 1999.
http://antares.in2p3.fr/Publications/conferences/2000/unreg_Lachartre_1999.pdf

[B-28] Motorola Semiconductor Products,
MPC860 : PowerQUICC™ Integrated PowerPC™ Microprocessor.
http://www.motorola.com/SPS/RISC/smartnetworks/products/intcomm/MPC860_M98655.htm

[B-29] The ATM Forum,
ATM Standardization.
http://www.atmforum.com/techspecfs1.html

[B-30] R. Gurin, A. Maslennikov,
ControHost Distributed Data Handling Package.
http://www.nikhef.nl/~ruud/HTML/choo_manual.html

[B-31] I. Pyarali, C. O'Ryan, D. C. Schmidt, N. Wang, V. Kachroo, A. Gokhale,
Applying Optimization Principle Patterns to Real-time ORBs, présenté à "COOTS '99" (5th USENIX Conference on OO Technologies and Systems, San Diego, California, USA), mai 1999.
http://www.usenix.org/publications/library/proceedings/coots99/full_papers/pyarali/pyarali.pdf

[B-32] J. Miller, J. Mukerji et al. (OMG Architecture Board ORSMC),
Model Driven Architecture (MDA), juillet 2001.
http://www.omg.org/cgi-bin/doc?ormsc/01-07-01.pdf

[B-33] R. Soley, OMG Staff Strategy Group,
Model Driven Architecture, draft 3.2, novembre 2000.
ftp://ftp.omg.org/pub/docs/omg/00-11-05.pdf

[B-34] Object Management Group,
OMG IDL Syntax and Semantics,
http://www.omg.org/cgi-bin/doc?formal/01-12-07.pdf

[B-35] Object Management Group,
Meta Object Facility (MOF) Specification - (version 1.3), mars 2000.
http://www.omg.org/cgi-bin/doc?formal/00-04-03.pdf

[B-36] T. Minka,
Software Patterns - IAP'97, mars 2000.
http://www-white.media.mit.edu/~tpminka/patterns/

Références C: Documentation papier

[C-1] S. Anvar, M. Mur,
The NA48 charged trigger: a high rate, high precision multiprocessor system,
"Proceedings of the International Conference on Computing in High Energy Physics", pp. 243 à 246, décembre 1992.

[C-2] G. Fischer, C. Avanzini, G.D. Barr, P. Calafiura, M. Cirilli, F. Costantini *et al*.
A 40 MHz pipelined trigger for $K^0 \rightarrow 2\Pi^o$ decays for the CERN NA48 experiment, in Nucl. Instr. Meth. A, vol. 419, pp. 623 à 631, Elsevier 1998.

[C-3] S. Anvar, F. Bugeon, P. Debu, J.L. Fallou, H. Le Provost, F. Louis, *et al*.
The charged trigger system of NA48 at CERN, in Nucl. Instr. Meth. A,
vol. 419, pp. 686 à 694, Elsevier 1998.

[C-4] D. Calvet, R. Hubbard, M. Huet, P. Le Dû, I. Mandjavidze, B. Thooris *et al.*,
Operation and performance of an ATM based demonstrator for the sequential option of the ATLAS trigger, in "Proceeding of the Xth IEEE Real Time Conference", pp. 101 à 108, 1997.

[C-5] S. Anvar, H. Le Provost, F. Louis,
The ANTARES offshore data acquisition: a highly distributed, embedded and COTS-based system, in "Proceedings of the 2000 IEEE Nuclear Science Symposium and Medical Imaging Conference", avril 2001.

[C-6] S. Anvar, F. Terrier,
A UML-based design process for distributed data acquisition and triggering systems in high energy physics experiments, in "IEEE Trans. on Nucl. Sc., part I, pp. 586 à 594, num. 3, vol. 48, juin 2001.

[C-7] J. Favaro,
Value-based prinicples for management of reuse in the enterprise, in "Proceedings of the Fourth International Conference on Software Reuse", pp. 23 à 26, avril 1996.

[C-8] D. Hamu, M. Fayad,
Achieving bottom-line improvements with enterprise frameworks, in "Communications of the ACM", vol. 41, n° 8, août 1998.

[C-9] D. Schmidt, M. Fayad,
Lessons Learned: Building Reusable OO Frameworks for Distributed Software, in "Communications of the ACM", vol. 40, n° 10, octobre 1997.

[C-10] M. Fayad, D. Schmidt,
Object-Oriented Application Frameworks, in "Communications of the ACM", Vol. 40, n° 10, octobre 1997.

[C-11] R. G. Lavender, D. C. Schmidt,
Active Object: an object behavioral pattern for concurrent programming,
in "Pattern Languages of Program Design 2", dirigé par J. M. Vlissides *et al.*,
pp. 483 à 499, Addison-Wesley, 1996.

Références C: Documentation papier

[C-12] D. C. Schmidt,
Acceptor and Connector, in "Pattern Languages of Program Design 3", dirigé par R. Martin *et al.*, pp. 191 à 229, Addison-Wesley, 1998.

[C-13] T. Quatrani,
"Visual modeling with Rational Rose 2000 and UML", Addison-Wesley, 1999.

[C-14] A. Lanusse, S. Gérard, F. Terrier,
Real-time modeling with UML: the ACCORD approach, in "Lecture Notes in Computer Science", vol. 1618 pp. 319 à 335, 1999.

[C-15] S. Gérard,
Modélisation UML exécutable pour les systèmes embarqués de l'automobile, Rapport de thèse de doctorat en Sciences, Université d'Evry, France, octobre 2000.

[C-16] J. Bosch,
Specifying frameworks and design patterns as architectural fragments, in "Proceedings of Technology of Object-Oriented Languages and Systems ASIA'98", pp. 268 à 277, juillet 1998.

[C-17] J. Bosch, P. Molin, M. Mattsson, P. Bengtsson,
Object-oriented framework-based software development: problems and experiences, in "ACM Computing Surveys", vol. 32, n° 1es, mars 2000.

[C-18] M. Mattsson, J. Bosch,
Framework Composition: Problems, Causes and Solutions, in "Proceedings of TOOLS USA'97", vol. 23, IEEE Computer Society, 1997.

[C-19] M. D. Lubars, N. Iscoe,
Frameworks Versus Libraries: A Dichotomy of Reuse Strategies, in "Proceedings of WISR'93, 6th Annual Workshop on Software Reuse", Owego New York, novembre 1993.

[C-20] W. Hasselbring,
Design of a communication framework for interoperable information systems, in "Proceedings of the Third World Conference on Integrated Design & Process Technology (IDPT'98)", Berlin, juillet 1998.

[C-21] D. Roberts, R. Johnson,
Patterns for evolving frameworks, in "Pattern Languages of Program Design 3", dirigé par R. Martin et al., pp. 471 à 486, Addison-Wesley, 1998.

[C-22] J. Coplien, D. Schmidt,
Frameworks and components, in "Pattern Languages of Program Design", dirigé par J. Coplien et D. Schmidt., pp. 1 à 5, Addison-Wesley, 1995.

[C-23] J. Viljmaa,
Tools Supporting the Use of Design Patterns in Frameworks, Rapport C-1997-25, Department of Computer Science, P.O. Box 26 (Teollisuuskatu 23) FIN-00014 University of Helsinki, FINLAND, mars 1997.

Références C: Documentation papier

[C-24] J.-P. Briot, R. Guerraoui,
Objets pour la programmation parallèle et répartie: intérêt, évolutions et tendances, in "Technique et science informatique", vol. 15, No. 6, 1996.

[C-25] R. Johnson,
Frameworks = (components + patterns), in "Communications of the ACM" Vol. 40, n° 10, octobre 1997.

[C-26] Y. Honda, M. Tokoro,
Soft real-time programming through reflection, in "Proceedings of the International Workshop on New Models for Software Architecture: Reflection and Metalevel Architecture", pp. 12 à 23, 1992.

[C-27] T. Watanabe, A. Yonezawa,
Reflection in an object-oriented concurrent language, in "Proceedings of Object-Oriented Programming Systems, Languages and Applications, 1988", ACM Press, septembre 1988.

[C-28] B.W. Boehm,
A spiral model of software development and enhancement, in "IEEE Computer", vol. 21, pp. 61 à 72, mai 1988.

[C-29] Wayne C. Lim,
Managing Software Re-Use 1/e, Prentice Hall, juin 1998.

[C-30] The ATM Forum,
ATM User-Network Interface Specification, Prentice-Hall, 1993.

[C-31] D. Calvet,
Réseau à multiplexage statistique pour les systèmes de sélection et de reconstruction d'événements dans les expériences de physique des hautes énergies,
Rapport de thèse de doctorat en Sciences, spécialité Electronique, Université Paris Sud, centre d'Orsay, France, mars 2000.

[C-32] P. Kruchten,
The Rational Unified Process: An Introduction, 2me édition, Addison-Wesley, 2000.

[C-33] D. Kulak, E. Guiney, E. Lavkulich,
Use Cases: Requirements in Context, Addison-Wesley, 2000.

[C-34] P. J. Ashenden,
The Designer's Guide to VHDL, 2me édition, Morgan Kaufmann Publishers, 2001.

[C-35] S. Prata,
C++ Primer Plus, 3me édition, Sams Publishing, 1998.

[C-36] Référence appel asynch & "Future" dans ACCORD

[C-37] C. Alexander, S. Ishikawa, M. Silverstein, M. Jacobson, *et al.*,
A Pattern Language: Towns, Buildings, Construction, Oxford University Press, 1977.

Références C: Documentation papier

[C-38] K. Beck, W. Cunningham,
Using pattern languages for object-oriented programs, Technical report, Tektronix, Inc., 1987. Présenté à OOPSLA-87, Workshop on Specification and Design for Object-Oriented Programming.

[C-39] E. Gamma, R. Helm, R. Johnson, J. Vlissides,
Design Patterns, Elements of Reusable Objet-Oriented Software, Addison-Wesley, décembre 1995.

[C-40] F. Buschmann, R. Meunier,
A System of Patterns, in "Pattern Languages of Program Design", dirigé par J. Coplien et D. Schmidt., pp. 325 à 343, Addison-Wesley, 1995.

[C-41] G. Meszaros, J. Doble,
A Pattern Language for Pattern Writing, in "Pattern Languages of Program Design 3", dirigé par R. Martin *et al.*, pp. 529 à 574, Addison-Wesley, 1998.

[C-42] J. Bézivin,
From Object Composition to Model Transformation with the MDA, in "Proceedings of TOOLS'USA", vol. IEEE TOOLS-39, Santa Barbara, août 2001.

[C-43] T. Reenskaug, P. Wold, O. A. Lehne.
Working with Objects, the OOram Software Engineering Method, Manning Publications, 1996.

[C-44] G. Sunyé, A. Le Guennec, J. M. Jézéquel,
Design Patterns Application in UML, in "Lecture Notes in Computer Science", vol. 1850, pp. 44 à 62, avril 2000

VIII

TERMES, ACRONYMES ET ABRÉVIATIONS

ADL	De l'anglais « *Architectural Description Language* ». Désigne toute une classe de langages – encore confinés au domaine de la recherche – consacrés à la description explicite des *architectures* logicielles (Cf. par exemple [C-16])
AGL	Atelier de Génie Logiciel. Désigne toute application destinée à fournir un environnement de développement et procédés de haut niveau pour l'analyse, la conception, et la réalisation d'applications.
API	De l'anglais « *Application Programming Interface* ». Ce terme désigne l'interface publique d'une librairie logicielle : une API est l'ensemble des signatures des fonctions d'une librairie ainsi que leur fonctionnalités. Par abus de langage, le terme API recouvre parfois également le code binaire de la librairie.
ASIC	Puce électronique spécialisée destinée à une application précise concue par ou à la demande de l'utilisateur.
Assemblage d'événement	En anglais « *Event building* » : opération typique d'un système TDAQ* consistant à regrouper une un seul paquet les données d'un même événement physique, au départ réparties sur tout le détecteur. Ce regroupement peut se faire de manière étagée en plusieurs étapes.
ATM	De l'anglais « *Asynchronous Transfer Mode* ». Norme d'interconnection concernant les couches physique et réseau, comprenant une norme à 155 Mb/s et une norme à 655 Mb/s. Les spécification de la norme concernent aussi bien le port ATM que la commutation. Contrairement à Ethernet*, ATM intègre la gestion de qualité de service (par exemple la réservation de bande passante) (Cf. [B-29], [C-30]).
Canevas	En anglais « *Framework* » : outil informatique de développement logiciel mettant en œuvre une architecture ainsi que des composants logiciels orientés vers un domaine d'application particulier.
CEA	Commissariat à l'Energie Atomique.
CERN	Organisation Européenne pour la Recherche Nucléaire, anciennement Centre Européen de Recherche Nucléaire. L'acronyme a été maintenu en dépit du changement de nom en raison de sa notoriété (Cf. [A-1])

Collaboration	On désigne ainsi par l'expression « la collaboration* », la communauté des laboratoires et instituts participant à une expérience
CompactPCI	Norme électronique mettant en œuvre la norme PCI* sur un bus de châssis.
CORBA	De l'anglais « *Common Object Request Broker* ». Norme définie par le consortium OMG* spécifiant un langage de description d'interface (IDL) et sa correspondance avec les langages à objets les plus répandus (C++ et Java) ainsi qu'une API* et des formats de communication à travers un réseau basés sur la notion d'ORB* et destinés à la réalisation et au déploiement d'applications distribuées orientées objet.
COTS	De l'anglais « *Commercial Off-The-Shelf* » ou « *Commodity Off-The-Shelf* ». Les expressions n'ont pas tout-à-fait le même sens : la première qualifie tout produit industriel matériel ou logiciel commercialement disponible sur le marché. La deuxième exige que le produit en question appartienne à un marché de masse lui assurant une certaine perennité et un coût très bas.
DAPNIA	Département d'Astrophysique, physique des Particules, physique Nucléaire et Instrumentation Associée. C'est le département d'accueil de cette thèse. Il fait partie de la Direction des Sciences de la matière du CEA*.
DCOM	De l'anglais « *Distributed Component Object Model* ». Norme analogue à CORBA*, définie par la société Microsoft.
Déclenchement	En anglais « *Trigger* » : terme spécifique à la physique des hautes énergies désignant un sous-système de traitement en temps réel sur un détecteur, permettant de réduire le flot de données par élimination en ligne des données non signifiantes.
Deadlock	Voir « Interblocage »
Design Pattern	Voir « Motif récurrent »

DSP	De l'anglais « *Digital Signal Processor* ». Processeurs spécialisé dans les calculs de traitement du signal en temps réel. Outre leurs unités et instruction de calcul spécialisées (par exemple pour les FFT), les DSP comprennent également des capacités d'entrées/sorties bien plus grandes que les microprocesseurs classsiques (liens de communication haut débit multiples, ports mémoire multiples, DMA multiples, etc.).
Ethernet	Norme d'interconnection concernant les couches « physique » et « liaison de données » comprenant une norme à 10 Mb/s, une norme à 100 Mb/s (*Fast Ethernet*) et une à 1 Gb/s (*Gigabit Ethernet*). Contrairement à ATM*, Ethernet n'intègre pas (ou presque) la gestion de qualité de service qui doit donc s'effectuer au niveau des couches plus hautes (Réseau ou plus).
Etiquette	En anglais « *Tagged Value* ». La norme UML* définit l'« étiquette » comme une spécification, définie par UML ou l'utilisateur, à base de paramètres susceptibles de prendre diverses valeurs. C'est, avec les stéréotypes*, l'un des mécanismes d'extension proposés par UML*. Sur un diagramme UML*, une étiquette se note {paramètre=valeur}.
Event Building	Voir « Assemblage d'événement ».
Fast Ethernet	Voir « Ethernet »
Firmware	Ce terme qualifie toute électronique implémentée sur des composants reprogrammables de type FPGA*. Il semble qu'il n'existe aucun terme officiel français équivalent.
FPGA	De l'anglais « *Fast Programmable Gate Arrays* ». Désigne les composants électroniques (re)programmables à volonté permettant de remplacer la conception de cartes à base de nombreux composants discrets par la conception de circuits à l'intérieur d'un petit nombre de puces. L'usage de FPGA pour la réalisation des cartes électroniques permet une réduction drastique du nombre de prototypes et du temps de déverminage. Le langage de programmation standard des FPGA est VHDL* (et, dans une moindre mesure Verilog)
Framework	Voir « Canevas »

Gigabit Ethernet	Voir « Ethernet »
HEP	De l'anglais « *High Energy Physics* »). Physique des Hautes Energies. Cette abréviation qualifie toutes les activités autour des expériences de physique nucléaire, de physique des particules et d'astroparticules.
IDL	De l'anglais « *Interface Definition Language* ». Langage de description d'interfaces objets normalisé par l'OMG* dans le cadre de la norme CORBA*.
Implémentation	Nous avons choisi d'utiliser cet anglicisme (accepté dans les dictionnaires récents, voir par exemple le dictionnaire anglais-français Robert & Collins) plutôt que les termes d'« implantation » ou de « mise en œuvre » qui correspondent à d'autres nuances. L'« implémentation » d'un système consiste en sa réalisation pratique sur une plate-forme particulière. Elle correspond à l'actualisation d'une architecture abstraite potentielle.
Interblocage	En anglais « *deadlock* ». Désigne tout blocage algorithmique dû a deux tâches concurrentes (ou plus) en attente d'une réponse de l'autre avant de poursuivre leurs traitements. Typiquement : A attend la réponse de B avant de lui donner sa réponse et B attend celle de A avant de donner la sienne.
LEP	De l'anglais « *Large Electron Positron collider* ». Collisionneur d'électrons-positron du CERN qui a accueilli les expériences les plus importantes du CERN dans les années 1980-1990.
LHC	De l'anglais « *Large Hadron Collider* ». Collisionneur de protons-antiprotons du CERN qui va servir les expériences de la prochaine génération (ATLAS, CMS, ALICE et LHCb). Il utilise le même tunnel de 28 km de circonférence du LEP* afin de préserver les investissements du LEP en travaux publics.
LIST	Laboratoire d'Intégration des Systèmes et des Technologies. Laboratoire de la Direction de la Recherche Technologique du CEA*.

Middleware	Ensemble de composants et – éventuellement – d'exécutifs logiciels destinés à fabriquer des applications de haut niveau. Le terme équivalent officiel français est « logiciel médiateur » mais sa connotation déterminée (par rapport au mot anglais qui désigne plutôt un *type* de logiciel) le rend impraticable.
MDA	De l'anglais « *Model Driven Architecture* », nouveau paradigme développé par l'OMG* consistant à prôner un développement « orienté modèles » des applications basé sur une spécification commune et normalisée des méta-modèles.
MORDICUS	Méthode à Objets Répartis de Développement Incrémental et de Conception UML* de Systèmes.
Motif récurrent	En anglais « *Design Pattern* » : structure et modèle d'implémentation logicielle typique constituant une solution à une problématique logicielle récurrente. Un motif récurrent et l'expression d'une solution éprouvée et expérimentée à un problème de conception logicielle. Elle a pour but de proposer une solution fondée sur des critères d'intelligibilité, de maintenabilité et d'élégance.
MTBF	De l'anglais « *Mean Time Before Failure* » : le MTBF d'un composant ou d'un module électronique est le temps moyen statistique entre deux pannes du module, tant que le module n'a pas epuisé sa durée de vie. Le MTBF est une mesure de la fiabilité d'un module.
OMG	De l'anglais « *Object Management Group* ». Consortium comprenant des sociétés et des établissement de recherche dont le but est la normalisation et la spécifications des technologies orientée objets, notamment dans un but d'interopérabilité entre *middlewares**.
ORB	De l'anglais « *Object Request Broker* ». Un ORB est un objet informatique jouant le rôle d'un intermédiaire entre un objet client et un objet serveur à travers un réseau, de sorte que le client n'ait pas à connaître la situation du serveur sur le réseau pour pouvoir lui transmettre ses requêtes et en obtenir les résultats.

PCI	De l'anglais « *Peripheral Component Interconnect* ». Norme électronique pour un bus d'interconnection rapide entre périphériques développée par Intel. Le PCI est un standard de fait dont le protocole électrique a été également mis en œuvre sous d'autres facteurs de forme industriels tels le chassis CompactPCI* ou les cartes filles PMC*.
Photomultiplicateur	Instrument très employé en physique des hautes énergies permettant de transformer le signal lumineux en signal électrique dont la charge est fonction de l'énergie déposée par les photons. Abréviation : PM*.
PM	Abréviation de « PhotoMultiplicateur »*.
PMC	De l'anglais « *PCI Mezzanine Card* ». Norme électronique de cartes mezzanine (ou cartes filles) mettant en œuvre sous ce facteur de forme la norme électrique PCI*.
RISC	De l'anglais « *Reduced Instruction Set Computer* ». Désigne une génération de microprocesseurs apparue au milieu des années 1980 dont l'architecture est basée sur un nombre réduit d'instructions par opposition aux microprocesseurs classiques comprenant un grand nombre d'instructions (CISC), et ce, pour permettre une rapidité d'exécution plus grande et une densité de transistors réduite. En fait, les processeurs RISC d'aujourd'hui sont aussi complexes que les CISC mais la terminologie s'est maintenue, de sorte que le terme a pris le sens de « processeur généraliste » par opposition à des processeurs plus spécialisés comme les DSP*.
RUP	De l'anglais « *Rational Unified Process* ». Méthodologie de conception / réalisation logicielles développée par la société Rational, basée sur la notation UML* et issue d'une synthèse entre les méthodologies objets développées par Booch, Rumbaugh et Jacobson (Cf. [C-33]).
SEDI	Service d'Electronique, des Détecteurs et d'Informatique du DAPNIA*. Laboratoire d'accueil de cette thèse.

SDRAM	De l'anglais « *Synchronous Dynamic Random Access Memory* ». Mémoire vive à accès synchrone. Ce type de mémoire est aujourd'hui (2001) la moins chère et la plus répandue tout en présentant de très bonnes performances (jusqu'à 133 MHz).
Stéréotype	En anglais « *stereotype* ». Spécification réduite à un seul mot clé, défini par UML ou par l'utilisateur, permettant de préciser le type d'un élément de modèle. Sur les diagrammes UML, un stéréotype se note entre guillemets français à proximité de l'élément concerné.
Tagged value	Voir « Etiquette ».
TDAQ	De l'anglais « *Trigger & Data Acquisition* » : en physique des hautes énergies, fait référence aux sous-systèmes d'acquisition et/ou de déclenchement* des détecteurs de physique.
Top-down	Qualifie toute approche de développement qui commence par des spécifications générales puis les précise progressivement, de manière à n'aboutir qu'à la fin du processus à la spécification détaillée et « bas niveau » du système.
Trigger	Voir « Déclenchement »
UML	De l'anglais « *Unified Modeling Language* ». UML est un langage de modélisation orientée objets destiné à la conception et spécification d'architectures et de systèmes logiciels complexes. UML comprend une sémantique et une notation graphique normalisées par l'OMG [B-11]. C'est un standard de fait de l'industrie logicielle.
Versioning	Voir « versionnement »
Versionnement	Néologisme. En anglais : *versioning*. Désigne la gestion des différentes versions d'un logiciel ou d'un système en général.
VHDL	Langage de conception électronique standard le plus répandu aujourd'hui dans l'industrie électronique. Ce langage intègre les notions d'interface et de composant mais non celle de classe (il ne connaît notamment pas les notions d'héritage et de spécialisation/généralisation).

VME	Abréviation de « *Versa Module Eurocard* ». Désigne une norme électronique de bus de châssis 32 bits développée par les sociétés Motorola, Signetics, Mostek et Thomson CSF. Cette norme est un standard largement utilisé pour des applications industrielles orientées temps réel. Le VME64 est une extension à 64 bits de la norme, assurant une compatibilité ascendante.

Résumé

La complexité croissante des systèmes d'acquisition et de traitement en temps réel (TDAQ) pour les expériences de physique des hautes énergies appelle à une évolution ad hoc des outils de développement. Dans cet ouvrage, nous traitons de l'articulation entre la spécification de principe des systèmes TDAQ et leur conception/réalisation sur une plateforme matérielle et logicielle concrète. Notre travail repose sur la définition d'une méthodologie de développement des systèmes TDAQ qui réponde aux problématiques de développement particulières à ces systèmes. Il en résulte la spécification détaillée d'un « canevas méthodologique » basé sur le langage UML, destiné à encadrer un processus de développement. L'usage de ce canevas méthodologique UML doit permettre la mise en place progressive d'un canevas « maison », c'est-à-dire un atelier de développement comprenant des composants réutilisables et des éléments d'architecture génériques adaptés aux applications TDAQ. L'ouvrage s'articule autour de 4 sections principales. La section II est dévolue à la caractérisation et à l'évolution des systèmes TDAQ. En section III, nous nous intéressons aux technologies pertinentes pour notre problématique, notamment les techniques de réutilisation logicielle telles les motifs récurrents (design patterns) et les canevas (frameworks), avec une orientation en faveur des domaines du temps réel et de l'embarqué. Notre apport conceptuel spécifique est exposé en section IV, où nous procédons notamment à la spécification détaillée, formalisée et exemples à l'appui, de notre modèle de développement. Enfin, nous terminons notre propos en section V en évoquant le projet de R&D MORDICUS de mise en œuvre concrète de notre canevas méthodologique UML, ainsi que les développements récents de l'OMG (Object Management Group) sur l'architecture orientée modèles (Model Driven Architecture), particulièrement en résonance avec notre travail.

Abstract

The increasing complexity of the real-time data acquisition and processing systems (TDAQ : the so-called Trigger and Data AcQuisition systems) in high energy physics calls for an appropriate evolution of development tools. This work is about the interplay between in principle specifications of TDAQ systems and their actual design and realization on a concrete hardware and software platform. The basis of our work is to define a *methodology for the development of TDAQ systems* that meets the specific demands for the development of such systems. The result is the detailed specification of a "*methodological framework*" based on the Unified Modeling Language (UML) and designed to manage a *development process*. The use of this UML-based methodological framework progressively leads to the setting up of a "home-made" framework, i.e. a development tool that comprises reusable components and generic architectural elements adapted to TDAQ systems. The main parts of this dissertation are sections II to IV. Section II is devoted to the characterization and evolution of TDAQ systems. In section III, we review the main technologies that are relevant to our problematic, namely software reuse techniques such as design patterns and frameworks, especially concerning the real-time and embedded systems domain. Our original conceptual contribution is presented in section IV, where we give a detailed, formalized and example-driven specification of our development model. Our final conclusions are presented in section V, where we present the MORDICUS project devoted to a concrete realization of our UML methodological framework, and the deep affinities between our work and the emerging "Model Driven Architecture" (MDA) paradigm developed by the Object Management Group.