CENTRE DE MAITRISE DES SYSTEMES ET DU LOGICIEL

Technologies pour la mise en œuvre des NCS

(*Network Centric Systems*)

⌘ ⌘ ⌘

L'interopérabilité des systèmes d'information et de communication et l'approche MDA

Application aux systèmes de gestion de crises

Jacques PRINTZ, Professeur au CNAM

Table des matières

Table des figures

1. PROBLEMATIQUE DE L'INTEROPERABILITE

Les systèmes d'information et de communication opérationnels

Le Ministère de la Défense distingue les systèmes d'information dits « opérationnels » des autres systèmes d'information jugés plus classiques, pour souligner le fait fondamental que ces systèmes sont utilisés en opérations (gestion des crises, maîtrise de la violence, maintien de l'ordre, opérations humanitaires, etc.) par les forces armées, la gendarmerie, les pompiers, les SAMU, etc.

Mener à bien une opération requiert de déployer différentes catégories de systèmes, chacun devant assurer une fonction qui lui est propre comme l'aide au commandement de haut niveau, le renseignement et l'observation, le pilotage des forces au plus près sur le terrain, la circulation, le soutien logistique, le soutien sanitaire, la communication avec la presse et/ou les structures locales, etc. Une opération comme celle du maintien de la paix au Kosovo met en branle de nombreux systèmes appartenant aux différentes nations contributrices : tous ces systèmes doivent, ou devraient, interopérer.

Ce qui est distinctif dans ces systèmes est que d'une part ils doivent être totalement autonomes (et mobile, de surcroît), ce qui interdit de partager physiquement des ressources, et que d'autre part ils doivent impérativement opérer de concert pour assurer la cohérence de la mission.

On peut schématiser cette situation comme suit :

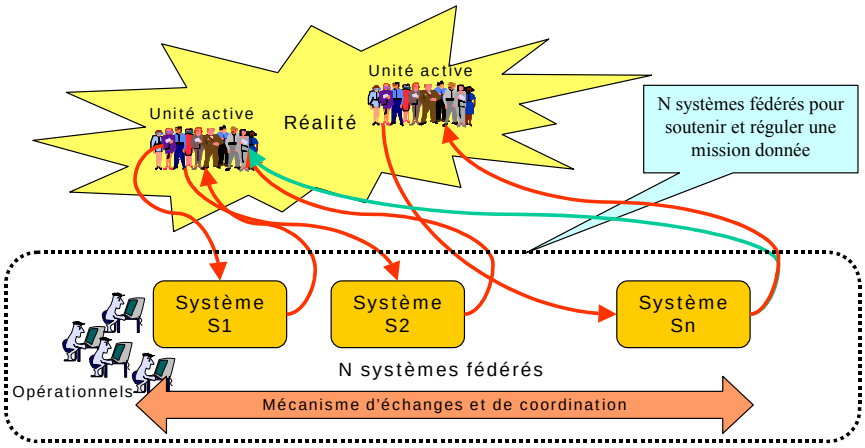


Figure 1.1 : Le pilotage des unités actives

Chaque unité active[1] sur le terrain est en interactions avec un ou plusieurs systèmes dont l'un pilote (« *control* » et « *command* », en anglais) effectivement, à l'instant t, l'unité active. Par définition, l'unité active est ce qui agit (« *perform* » en anglais, d'où la notion de performatif, introduite en linguistique par J.Austin[2]). Chacun des systèmes héberge un nombre d'acteurs « opérationnels » plus ou moins grands (pouvant atteindre quelques centaines) qui auront à prendre des décisions concernant les unités actives dans des délais qui peuvent être très courts, au vue de la situation générale et de l'état des différentes unités actives.

L'efficacité opérationnelle des unités actives requiert des mécanismes d'échanges d'information, et de coordination des actions conduites et/ou projetées au niveau des

[1] Sur la notion d'unité active, voir par exemple, Th. De Montbrial, *L'action et le système du monde*, Hermann. Dans la littérature NCW et C4ISR/DODAF, on parle de « battlefield entities » ; c'est la même chose.

[2] Cf. *How to do things with words*, traduit en français, au seuil : *Quand dire, c'est faire*.

systèmes, de telle sorte que ceux-ci apparaissent comme une fédération, c'est-à-dire un système unique parfaitement cohérent (c'est du moins l'objectif). Les systèmes jouent le rôle de régulateurs par rapport aux différents processus qui sont à l'œuvre dans la réalité.

La coordination des opérations, et la régulation globale, se fait par échanges de messages entre les systèmes et entre les unités actives. Il est donc primordial que les différentes communautés d'opérationnels, unies autour d'une même mission, comprennent le contenu des messages exactement de la même façon, indépendamment des métiers qui leur sont propres. Le résultat de ces interactions sera la construction d'une situation globale de la réalité, partagée entre tous les acteurs, régulièrement mise à jour, appelée COP (*Common Operational Picture*) dans la terminologie OTAN.

Si l'on considère deux systèmes en interaction, la systémique des échanges peut se représenter comme suit :

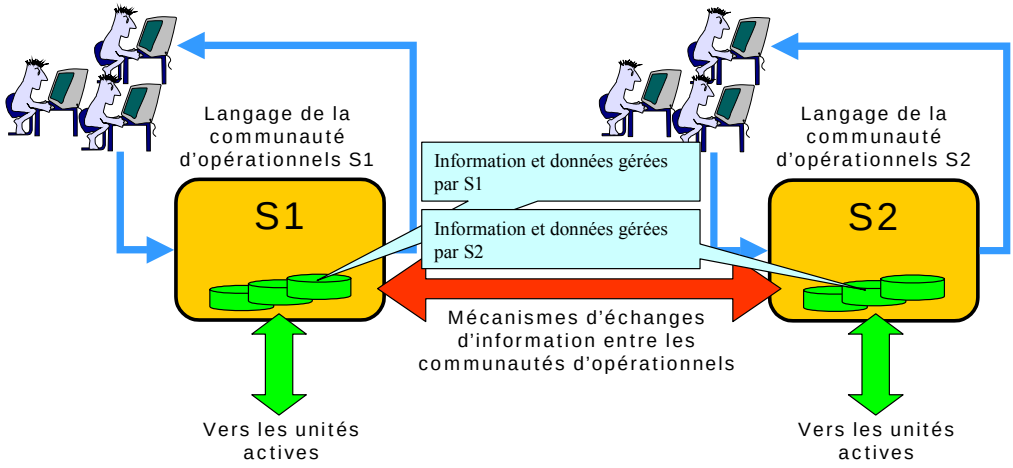


Figure 1.2 : Echanges entre les systèmes de pilotage

Chacun des systèmes gère les informations qui lui sont propres avec des moyens ad hoc : bases de données, espaces de travail publics et privés pour un ou plusieurs opérationnels, règles de gestion plus ou moins formalisées (i.e. des « workflow »), etc.

Dans de tels systèmes, on peut concevoir que la représentation des informations soit spécifique à chacun des systèmes, ne serait-ce que pour des raisons industrielles. Il est par contre impératif que la sémantique soit la même, faute de quoi il est inéluctable que les actions entreprises par les unités actives deviennent incohérentes. En plus des problèmes d'ingénierie difficiles que pose ces systèmes (grande complexité, grande taille autour du million de lignes source C++/Java ou 10 à 15.000 points de fonctions, IHM sophistiquées avec bibliothèques de symboles graphiques[3], cartographie, projets complexes nécessitant des interactions entres des centaines d'acteurs, etc.), il vient se greffer un problème nouveau qui est celui de l'ingénierie sémantique, qui certes existe toujours de façon plus ou moins explicite, mais qui prend ici une importance vitale pour la sécurité des échanges entre les unités actives qui participent aux opérations.

L'interopérabilité des systèmes S1, S2, …, Sn, présuppose que les communautés d'usagers de chacun d'entre eux ont des informations en communs susceptibles de s'échanger. De deux choses l'une : ou bien chacun négocie ses échanges avec tout le monde au cas par cas, ou bien les communautés d'usagers adoptent par convention et volontairement un langage commun pour faciliter les échanges qui sera le bien commun de tous sans appartenir à personne. Dans le premier cas, le nombre de dialogues

[3] Voir, par exemple, le document du DOD Mil-std-2525B, *Common warfighting symbology*.

possibles entre N systèmes est $N(N-1)$; c'est une complexité quadratique. Dans le second cas, il suffit de connaître le langage commun pour dialoguer avec tous ; c'est une complexité linéaire. Ce langage commun est appelé langage pivot. Son implémentation au sein des systèmes constitue l'architecture pivot. On examinera au chapitre 4 quelques unes des propriétés de ce langage et de l'architecture correspondante.

Remarque concernant la complexité combinatoire

La combinatoire des systèmes pris deux à deux conduit à un nombre de liaisons statiques égal à $\frac{N(N-1)}{2}$, d'où les $N(N-1)$ dialogues. En examinant la distribution des échanges dans le temps, tout au long d'un cycle d'utilisation des systèmes S1, S2, …, Sn, il faut également considérer toutes les parties possibles car la combinaison de deux peut influencer un tiers, et ainsi de suite. L'ensemble des interactions possibles est donc la somme des combinaisons 2 à 2, 3 à 3, etc. soit au total 2^N qui est une combinatoire exponentielle.

Le laissez-faire en matière d'interopérabilité conduit donc inéluctablement à une complexité exponentielle dont le coût de gestion devient rapidement prohibitif, du moins si l'on souhaite garantir un niveau de qualité durable.

Quelques définitions de l'interopérabilité

Définition proposée par l'ISO

La possibilité de communication, d'exécution de programmes ou de transfert de données entre unités fonctionnelles différentes, de telle manière que l'utilisateur n'ait que peu ou pas de besoin de connaître les caractéristiques propres à chaque unité.

Définition de l'interopérabilité du *Glossaire OTAN de termes et définitions français et anglais* AAP-6 (Edition 2002)

« Aptitude des forces de l'Alliance et, lorsqu'il y a lieu, de forces des pays partenaires et d'autres à s'entraîner, à s'exercer et à opérer efficacement ensemble en vue d'exécuter les missions et les tâches qui leur sont confiées ».

Définition proposée par le DOD

« The ability of systems, units, or forces to provide services and accept services from other systems, units, or forces and to use the services so exchanged to enable them to operate effectively together ».

Ces deux dernières définitions mettent bien en évidence l'intégration des unités actives dans la problématique de l'interopérabilité, indépendamment des aspects techniques propres à l'informatique. L'interopérabilité concerne la globalité du système d'information, acteurs compris.

2. RENDRE LES SYSTEMES INTEROPERABLES

Aperçu sur la sémantique

Pour que deux ou *N* systèmes soient interopérables au sens des définitions ci-dessus, il faut que les données échangées soient interprétées de la même façon par chacune des communautés d'opérationnels. Ceci nous amène à devoir clarifier rigoureusement le sens de la donnée et la représentation de la donnée. Pour un opérationnel qui s'apprête à décider quelque chose, ce qui est fondamental c'est le sens porté par les entités

informatiques visualisées sur son écran de contrôle par rapport à la réalité. En d'autres termes, ce qu'il « voit » à l'écran est-il « VRAI[4] » ?

La distinction des trois niveaux : syntaxe, sémantique et pragmatique, tels que proposée par la théorie de l'information[5] est ici tout à fait pertinente. L'important pour l'opérationnel c'est l'information, c'est-à-dire la somme des trois.

Ce qui est délicat avec la sémantique est que pour en parler il est indispensable d'utiliser une syntaxe. La sémantique nous renvoie, d'un point de vue phénoménologique, au concept même, c'est-à-dire ce qu'il faut avoir compris (i.e. fait sien) pour agir efficacement. Un concept peut s'exprimer de façon quantitative (i.e. un nombre, une forme géométrique), ou de façon qualitative (un texte libre plus ou moins structuré, une couleur, une criticité, un signe conventionnel, etc., c'est à dire un ensemble discret permettant de définir une échelle).

- ✓ Dans le premier cas, il faut un système d'unités, un typage comme on dit en informatique, pour interpréter correctement le nombre[6] qui la représente.
- ✓ Dans le second cas il faut que : a) les codes de représentation soient spécifiés de façon exhaustive, b) que les valeurs correspondantes soient données une fois pour toute, et ceci de façon explicite pour éviter les paradoxes logiques associés aux négations (l'ensemble doit être immuable).

Pour définir la sémantique dont nous avons besoin pour aborder l'interopérabilité, il convient de bien distinguer :

a) le problème de la référence, c'est à dire l'adressage des entités (NOMMER[7]), en distinguant clairement le nom du conteneur, et la valeur du contenu.
b) le problème des transformations ou changement d'état, c'est-à-dire des transactions effectuées (TRANSFORMER) ; la transformation est à envisager sous différents aspects, fonctionnels et non fonctionnels, pour lesquels nous adopterons, pour raison de commodité, la classification de la norme ISO/CEI 9126 résumée par l'acronyme FURPSE (Functionality, Usability, Reliability, Performance, Serviceability, Evolutivity).
c) le problème de l'action (DECIDER, c'est-à-dire envoyer des ordres [COMMAND] et contrôler, c'est-à-dire s'assurer de leur bonne exécution [CONTROL]).

Nous dirons que le sens d'une entité informatique manipulée et/ou échangée au sein d'une fédération de système (et à fortiori au sein d'un système, mais c'est plus simple !) est l'ensemble des règles qui définissent son usage, ***dans tout contexte où elle peut être utilisée***. La sémantique intègre les problématiques des différents acteurs qui ont affaire avec les entités informatiques, directement ou indirectement ; c'est le rôle du modèle d'entreprise d'effectuer cette collecte.

Le problème de la référence est celui de l'identifiant unique. Un même objet de la réalité ne peut (ne doit !) avoir qu'une seule identification pour garantir la cohérence des décisions des opérationnels et des actions effectuées par les unités actives.

Dans les ordinateurs, on fait un distinction fondamentale entre l'« adresse » du conteneur de l'information, i.e. son chemin d'accès, et le contenu informationnel, i.e. la valeur de ce contenu. Cette distinction est cruciale dans l'ingénierie des systèmes d'exploitation, des SGBD et des compilateurs qui doivent manipuler l'une ou l'autre de

---

4 Ceci renvoie à un texte profond de A.Tarski : *Le concept de vérité dans les langages formalisés*, dans *Logique, sémantique, métamathématique*, Tome 1, chez Armand Colin.

5 Cf. C. Shannon, W. Weaver, *The mathematical theory of communication*.

6 Cf. l'interprétation de l'altitude sur l'IHM de l'Airbus lors de la catastrophe du Mont Saint Odile.

7 Ceci renvoie aux travaux de G.Frege, en particulier la distinction entre sens et dénotation ; cf. *Ecrits logiques et philosophiques*, Seuil.

ces deux formes sans jamais les confondre, en particulier pour tout ce qui traite de l'optimisation (chemin d'accès et caches, hiérarchie de mémoires, optimisation des calculs, etc.).

Le problème des transformations opérées au sein d'un système (i.e. le « *business computing* ») est celui de la fidélité du modèle informatique par rapport à la réalité. Si tel stock de carburant a été consommé par une unité active, ceci doit être exactement reflété dans les bases de données des systèmes, faute de quoi il y aura encore incohérence des actions. Notons que cela pose le problème des actions irréversibles dans la réalité, car dans le modèle informatique, qui n'est qu'un monde virtuel, on peut toujours revenir en arrière.

Le concept sémantique sous-jacent est celui de transaction et de moniteur transactionnel qui ont fait l'objet de très nombreuses études, sur lesquelles nous ne reviendrons pas[8]. Notons que les transactions du monde réel sont toujours des transactions dites longues, par opposition aux transactions courtes des SGBD.

Le problème de l'action est celui du droit à faire-faire et du contrôle. Le système S1 donne un ordre à une unité active, ou à un autre système, de faire quelque chose, selon certaines modalités convenues. La sémantique de l'action implique de la part de l'émetteur un suivi des effets observés, et de la part du récepteur qui agit, la rédaction de compte-rendus décrivant la progression de l'action entreprise dans la réalité. Dans ce cas, il s'agit donc de contrôler l'exécution d'un processus distant.

Dans le cas de la figure 1.2 ci-dessus, cela revient à dire que S1 qui émet des ordres vers S2, pilote la mise en œuvre de processus, eux-mêmes contrôlés localement par S2. Entre les règles qui définissent les messages et ce qui s'échangent réellement, il y a la même différence qu'entre le langage et sa grammaire d'une part, et la langue utilisée par les usagers de ce langage d'autre part : l'usager peut commettre des erreurs ; cette distinction est fondamentale en linguistique depuis F. de Saussure[9].

La figure ci-dessous donne un exemple simplifié d'une telle sémantique (sémantique d'acquittement des messages).

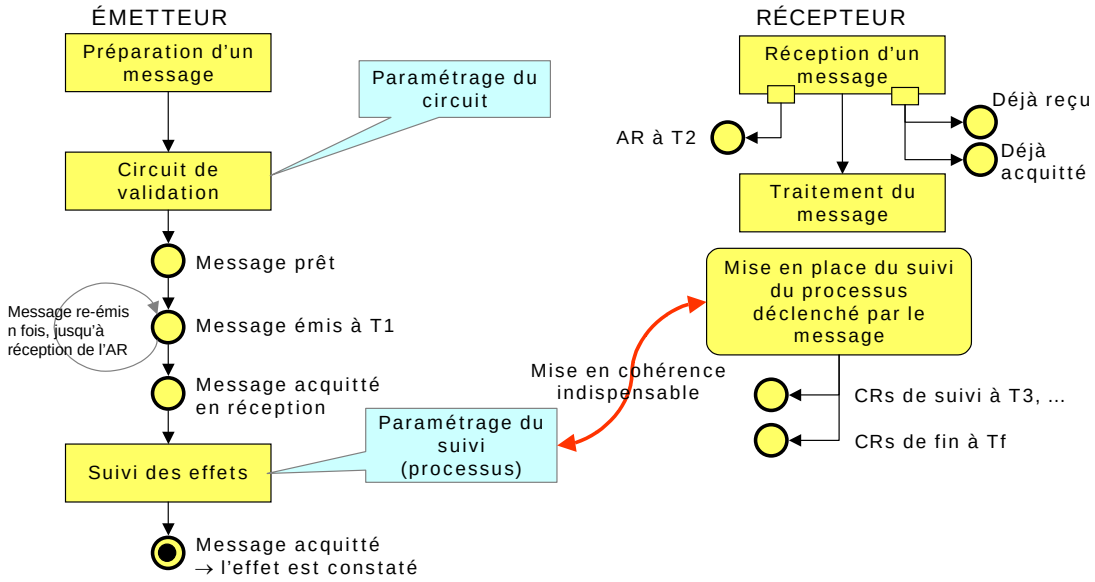


Figure 2.1 : Sémantique de l'acquittement d'un message

[8] La référence, dans ce domaine, est J.Gray, A.Reuter, *Transaction processing: concepts and techniques*, Kluwer.

[9] Cf. *Cours de linguistique générale*, Payot.

Dans ce cas, l'information échangée entre les systèmes S1 et S2 est un automate de contrôle qui doit préciser exactement qui fait quoi, là encore pour obtenir une cohérence effective des actions. S1 fixe un cadre conventionnel, négocié avec S2, dans lequel S2 inscrira son action, avec des modalités de détail spécifiques à S2 que S1 n'a pas — ne doit pas — connaître, ceci en application du principe de subsidiarité.
NB : On pourrait calculer en quantité d'information au sens de Shannon ce que coûterait une politique du type « tout le monde cause avec tout le monde ». Les événement rares au niveau de la fédération ont une quantité d'information élevée ; d'où la subsidiarité qui trouve ici sa justification théorique.

Les différents niveaux de modélisation et les relations entre modèles

Dans un monde idéal ou maîtres d'ouvrage et maîtres d'œuvre joueraient parfaitement leur rôle, le référentiel système devrait contenir différentes catégories de modèles, conformément aux différentes activités du processus de développement[10] dont ils constituent les données d'entrée ou les résultats.
Pour illustrer ces différentes étapes, en matière de données, nous aurons, au niveau métier : les modèles de flux ; au niveau conception générale : les modèles conceptuels de données (ce qui implique préalablement le découpage — c'est une relation de dualité — du système en en Applications, en Services et en Transactions, analogue à la hiérarchie paquetages, classes et méthodes en objet) ; au niveau conception détaillée, les modèles logiques écrits dans le DDL du SGBD choisit pour la réalisation ; au niveau programmation : les modèles physiques tels que les programmeurs les découvriront dans les différentes mémoires de la machine.
Enfin, tout ou partie de ces informations seront utilisées pour faire les tests d'intégration et la recette du système.
Concernant la partition de l'ensemble des données et de celui des traitements en applications et en transactions, il faut être conscient des relations très étroites et très subtiles qui existent entre les deux classifications, car le voisinage sémantique ne coïncide généralement pas avec la topologie du rangement en mémoire, d'où des performances parfois calamiteuses[11] lorsque l'on oublie ces relations de voisinage.
Le schéma ci-dessous montre le raffinement progressif des familles de modèles et leurs relations, jusqu'au stade ultime du déploiement.

10 Cf. les artefacts recommandés par le processus unifié (UP) de l'OMG ; ou ceux de la norme ISO/CEI 12207.

11 Cf. B.McNutt, *The fractal structure of data reference, Applications to the memory hierarchy*, Kluwer.

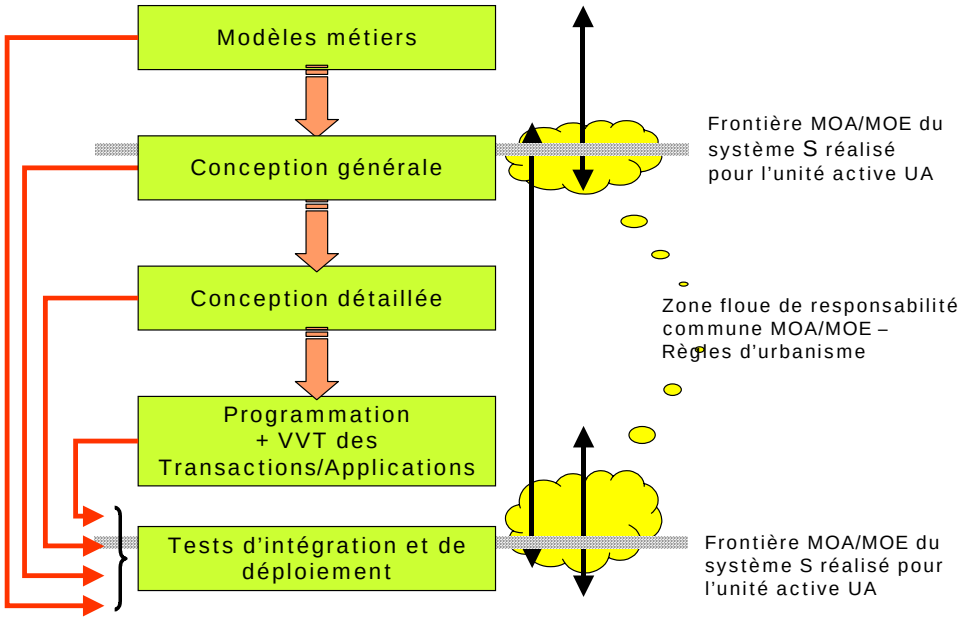


Figure 2.2 : Enchaînement des modèles

Dans le monde réel des projets informatiques, rien de tout cela n'existe, et il est très rare que la cohérence entre tous ces modèles ait été entretenue. En général seul l'étage CONCEPTION DETAILLEE – PROGRAMMATION fait foi, car in fine, c'est ce que la machine exécute.

De plus les frontières MAITRISE D'OUVRAGE – MAITRISE D'ŒUVRE sont floues[12]. D'une organisation de développement à l'autre, d'une entreprise à l'autre, ces frontières sont éminemment variables. Très souvent, les modèles métiers sont réduits à leur plus simple expression, à savoir des documents en langage naturel, manquant totalement de précision pour un usage informatique sérieux.

Le problème sous-jacent qu'il serait naïf de nier, et pire de brocarder, est celui de l'harmonisation de deux univers antagonistes, pour ne pas dire contradictoires.

Dans la partie métier, il est hors de question d'utiliser un langage formalisé, au sens informatique et/ou mathématique du terme. Les organisations sont des systèmes ouverts, qui doivent rester ouverts. Il est donc normal que le langage naturel, qui est encore le meilleur moyen de communiquer entre les humains, y règne en maître. Notons que le langage naturel n'interdit pas d'être rigoureux, et il y a fort à parier que quelqu'un qui n'est pas rigoureux, ne le sera jamais quel que soit le langage utilisé.

Dans la partie informatique, seuls les langages informatiques sont utilisés. Pour un système de type SIOC, avec des IHM, des bases de données, des réseaux,… le nombre de dialectes utilisés pourra dépasser la dizaine. Pour que le programmeur puisse agir efficacement sur la machine il est impératif que ces différents langages soient parfaitement définis, y compris, et surtout, en cas de fonctionnement anormal, lors des activités de validation, vérification et test. Les textes et programmes écrits dans ces différents dialectes devront être cohérents entre eux, pour que le processus informatique soit lui-même globalement cohérent. Rien n'est pire, pour un architecte de système, que les défaillances non reproductibles. Idéalement, le système devrait être déterministe, de façon à pouvoir reproduire tout type de pannes, et remonter ainsi aux erreurs humaines causes des défaillances.

[12] Sur ces sujets voir le site du club des maîtres d'ouvrage : www.clubmoa.asso.fr

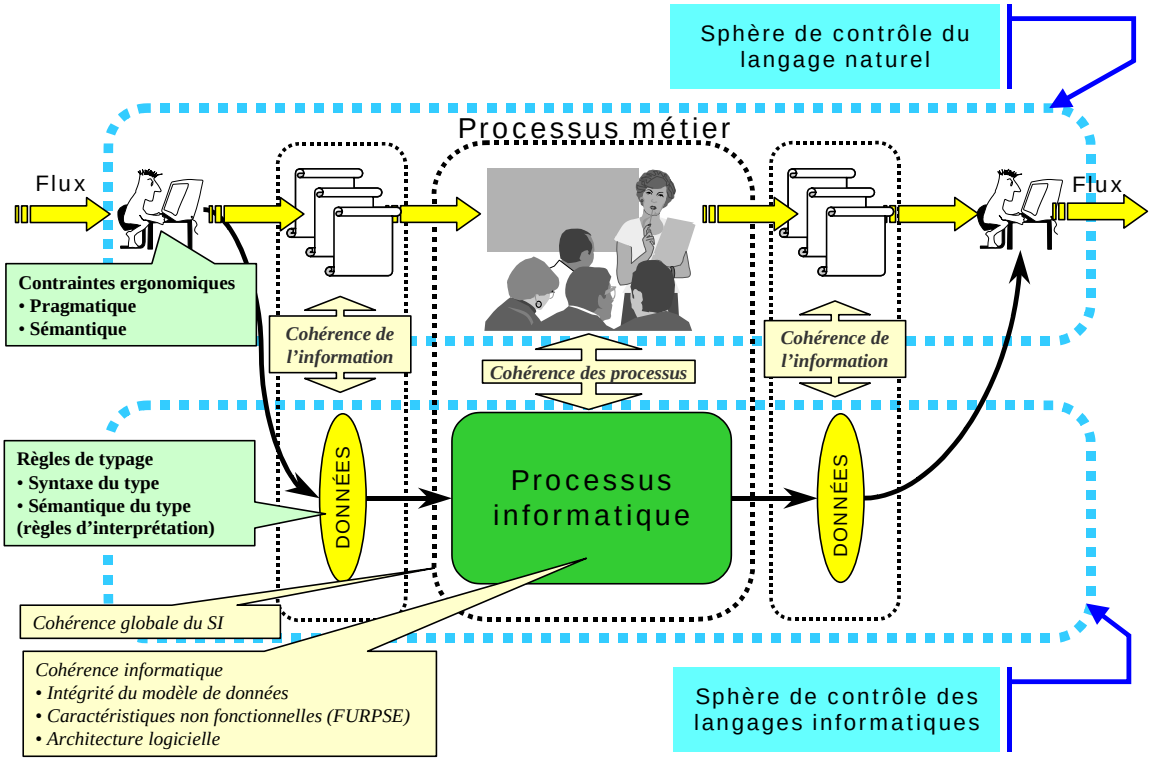


Figure 2.3 : Cohabitation langage naturel et langages informatiques

Le point dur de la modélisation est la traduction d'un univers ouvert dans l'univers fermé mais extensible, déterminisme oblige, qui est celui de l'informatique[13].
La problématique fondamentale est donc celle de l'évolutivité de l'ensemble des textes écrits dans les différents dialectes utilisés, et des modèles qui en définissent la logique d'assemblage, sans oublier les tests.
On pourrait édicter comme principe que la partie informatique du système est par essence inachevée, incomplète par nature, et que en conséquence, les caractéristiques les plus fondamentales dans FURPSE sont S et E, du moins pour des systèmes de systèmes de ce type. C'est une exigence architecturale très forte.
La solution du problème conditionne complètement la capacité d'évolutivité de la partie informatisée du système, par rapport aux exigences des métiers, qui elles ne dépendent que de la concurrence ou de nouvelles orientations stratégiques des différents métiers.
Notons que la partie informatisée du système a également sa propre dynamique évolutive du fait des évolutions technologiques.
En cas d'évolution du métier, il faut donc pouvoir retoucher, et si possible de façon automatique, l'ensemble des modèles et des textes qui réalisent le processus informatique.
L'impact des évolutions, en aménageant le schéma précédent, se manifeste à tous les niveaux de modélisation. La gestion des traçabilités des modifications qui en résultent est primordiale.

[13] Sur ces notions, voir : J.Printz, *Puissance et limites des systèmes informatisés*, Hermès.

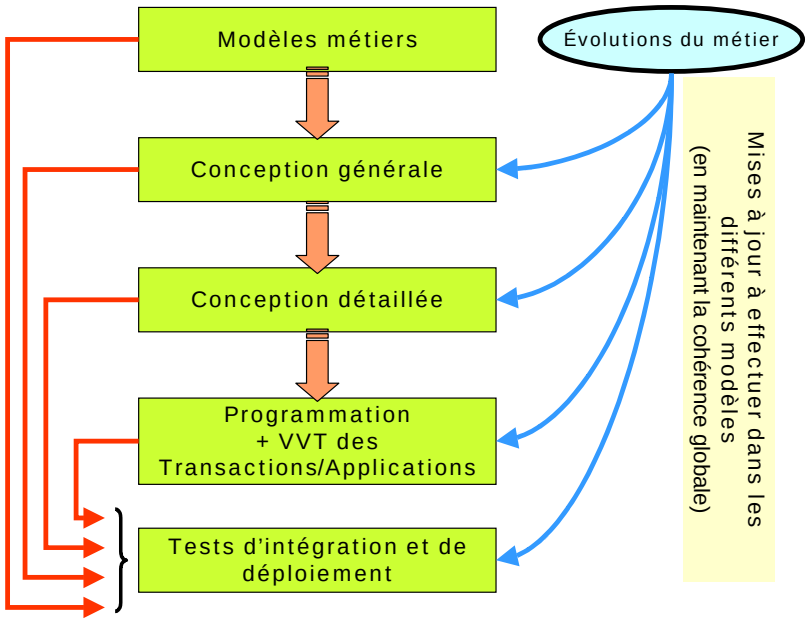


Figure 2.4 : Traçabilité et mise à jour des modèles

On peut concevoir que le processus de retouche, s'il n'est pas correctement géré, avec une méthodologie ad hoc, et raisonnablement automatisé, du moins dans ses parties les plus critiques, pourra avoir des délais de réalisation incompatibles avec la dynamique métier et les exigences qualité.
Dans ce cas, la sagesse est de ne pas informatiser.

Concept de fédération de systèmes

Dans la réalité, on part toujours d'un existant comportant N systèmes que l'on souhaite faire intéropérer, car il est économiquement impossible de tout refaire. Pour des raisons d'efficacité opérationnelle, il est nécessaire d'échanger des données par des moyens informatiques, voire même de partager des services, entre les différents systèmes qui constituent la fédération.
D'un point de vue informatique on souhaite arriver à une situation où chacun des systèmes a fait l'inventaire de ses ressources de façon à identifier ce qui pourrait être mis dans un pot commun, tant au niveau des données, que des services.

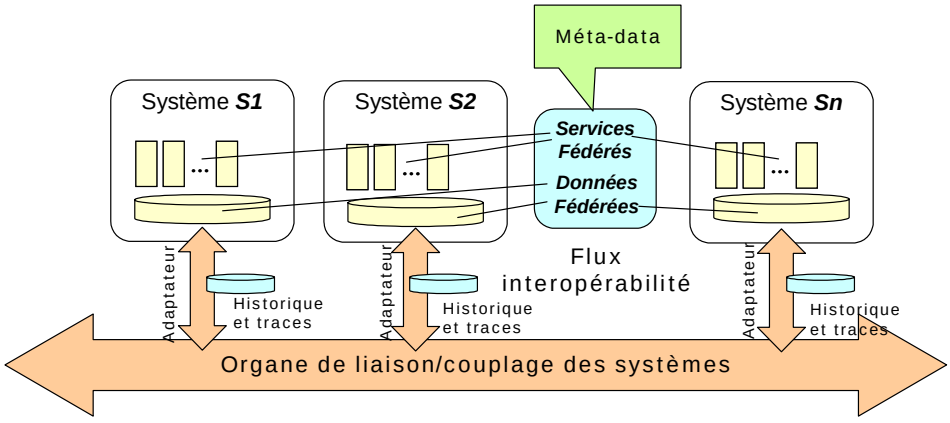


Figure 2.5 : Fédération de systèmes

Sont dits fédérés, des données et/ou des services qui peuvent être utilisés par d'autres systèmes que celui qui en est le propriétaire. Ce qui a été fédéré est soumis à des règles d'évolutivité plus contraignantes car toute évolution intempestive de ces entités pourra occasionner des incohérences chez ceux qui les utilisent. Services et données fédérés sont décrits à l'aide d'une méta-description. D'où le concept de chaînes de liaisons qui matérialisent les nouveaux niveaux de couplage obtenus du fait de l'interopérabilité.

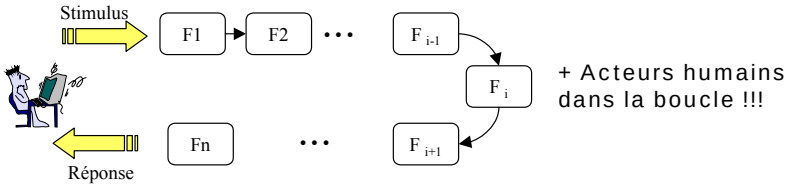


Figure 2.6 : Chaînes de liaisons dans une fédération

Un acteur opérationnel, d'un quelconque des systèmes de la fédération, peut engendrer une chaîne de liaison qui met en jeu différentes fonctions dans différents systèmes, ce qui amène deux nouveaux problèmes :

- ✓ La fiabilité de la chaîne de liaisons, et en particulier celle des entités fédérées, car la fiabilité globale F de la chaîne est le produit des fiabilités des fonctions qui la constitue ; on sait que F tend vers 0, en fonction de la longueur de la chaîne.
- ✓ La localisation des défauts en cas de défaillance et le recueil des événements correspondants qu'il faut pouvoir observer avant même de savoir les contrôler, d'où l'importance des fonctions de traces qui permettent d'historiser tout ce qui entre et sort de chacun des systèmes au titre de la fédération (donc de connaître parfaitement la sphère de contrôle de chacun des systèmes).

Rendre des systèmes interopérables revient donc à spécifier et à réaliser le schéma d'interconnexion et de couplage des différents systèmes, tant au niveau de la syntaxe et de la sémantique des échanges, ceci de façon durable pour tenir compte des évolutions asynchrones inévitables de chacun d'eux, et sans dégrader le contrat de service d'aucun d'entre eux.

L'aspect couplage est fondamental, car les couplages intempestifs sont la source des « canaux cachés », préjudiciables à la sécurité, à la fiabilité et à l'évolutivité. Il est commode de distinguer trois niveaux de couplage, d'intensité décroissante : le couplage par les traitements, le couplage par les données, le couplage par l'ordonnancement, c'est-à-dire ceux réalisés par les évènements qui déterminent l'ordonnancement.

NB : Pour plus de détails, voir l'Annexe 1, Nature des couplages.

Homogénéiser les niveaux de modélisation

Le premier problème auquel on se heurte, lorsque l'on souhaite rendre des systèmes interopérables, est celui de l'hétérogénéité du contenu des modèles, ceci indépendamment de la précision des informations contenues dans les modèles.

Il n'y a aucune chance pour que deux architectes, opérant dans deux systèmes distincts, aient les mêmes objectifs de modélisation, quand bien même ils utiliseraient les mêmes outils.

Pour simplifier l'exposé, considérons ce qu'il est nécessaire de faire sur les données échangées (ou échangeables) pour rendre les modèles correspondants comparables, dans le cas de deux systèmes.

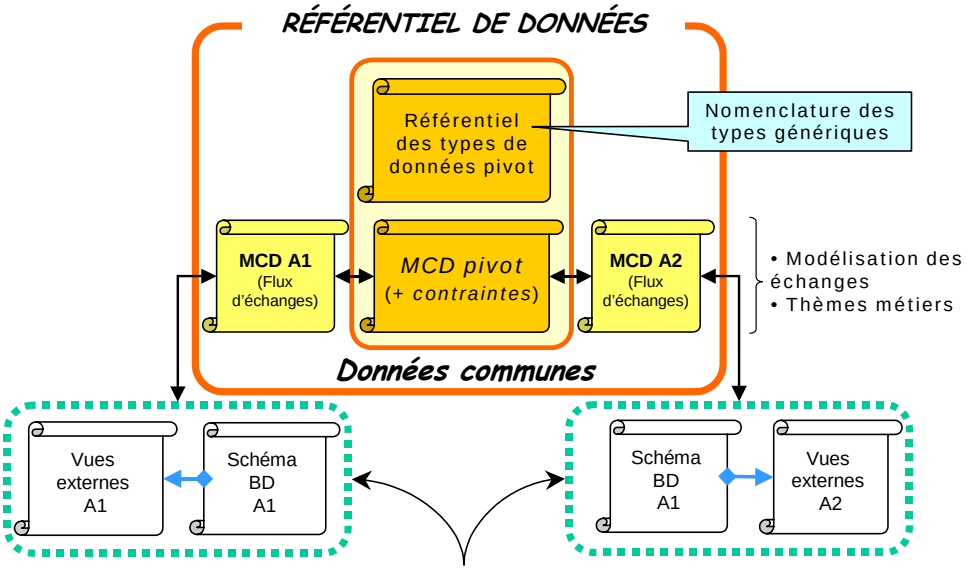


Figure 2.10 : Homogénéisation des modèles

Le principe de la reconstruction consiste à répartir des existants de chacun des systèmes et à fabriquer les MCD correspondant aux données échangées avec les outils de modélisation utilisés dans chacun des systèmes. S'il n'y a pas d'outil, on prendra, par défaut, le modèle de classes UML qui est conforme au modèle général ENTITE – RELATION – ATTRIBUT (ERA)[14].

A ce stade on dispose de deux MCD dont il faut valider la sémantique avec les acteurs métiers. Il faut donc mettre les MCD dans un format compréhensible par les acteurs métiers : c'est le rôle du MCD pivot.

Le MCD pivot est un format « neutre » qui ne privilégie aucun des systèmes de la fédération (pour des raisons évidentes de contraintes économiques et de contraintes de stratégie industrielle) ; c'est la partie données du langage pivot qui sera ébauché au chapitre 4. Valider que les MCD sont cohérents du point de vue de la sémantique, revient à s'assurer qu'ils peuvent être traduits les uns dans les autres, comme on pourrait le faire avec des textes écrits en langages naturels. Dans le cas du pivot, la traduction a un point fixe (ce qui n'est généralement pas le cas avec les langues naturelles, compte tenu du flou sémantique des langues naturelles).

Le rôle du référentiel de types est de caractériser de façon rigoureuse tous les attributs élémentaires utilisés dans le MCD pivot, et ceci tant du point de vue quantitatif, que du point de vue qualitatif. Ce référentiel de type est une nomenclature métier à partir de laquelle on va reconstruire les différents modèles (dans le jargon des dévots de l'objet, c'est ce qu'on appelle une ontologie). Les règles de partition pour construire la nomenclature sont celles de la logique et de la théorie des ensembles (il s'agit d'établir un ordre compréhensible et acceptable par tous les acteurs).

Le format neutre le plus naturel pour exprimer le MCD pivot est aujourd'hui le langage XML qui ne nécessite pas de compétence informatique particulière pour être compris ; la logique de ce langage est purement grammaticale et générative[15], d'où son

[14] Voir l'ouvrage de T.Teorey, *Database modeling and design*, Morgan Kaufmann.

[15] Cf. N.Ruwet, *Introduction à la grammaire générative*, Plon.

universalité. Il est légitime de parler de type syntaxique pour ce mode de représentation qui est celui du web[16].
Des langages comme ASN.1[17] ont joué le même rôle dans le domaine des messageries X.200, sauf que, en matière de protocole, il faut spécifier l'échange au niveau du bit. Mais formellement, c'est la même problématique.
Le schéma ci-dessous résume l'ensemble de la démarche de rétro-conception (i.e. le *refactoring*).

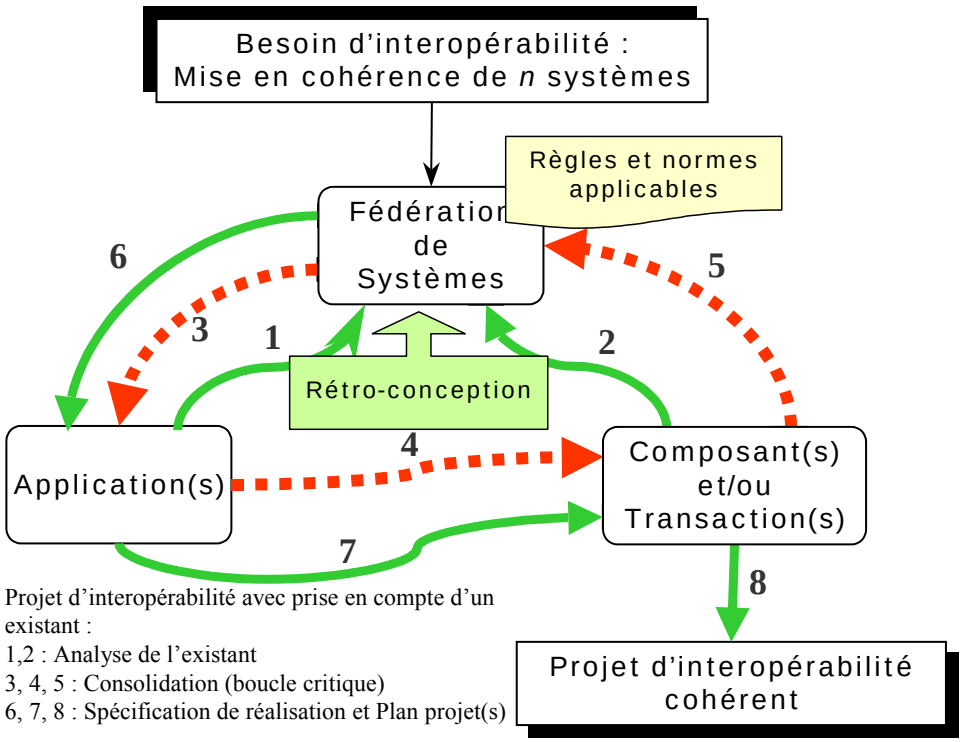


Figure 2.11 : Démarche de mise en cohérence des modèles

On remarquera que la méthode proposée exige de visiter deux fois le niveau fédération, ceci afin de prendre en compte la non fidélité des MCD initiaux vis-à-vis de l'implémentation.
Le chemin 6, 7, 8 doit être un automatisme comme indiqué sur la figure 2.12.

Le problème des données fédérées

Du point de vue de la modélisation des données, les données fédérées sont des généralisations strictes des données correspondantes au sein des différents systèmes S1, S2, … Sn ; ces modèles doivent être indépendants des plates-formes. Inversement, les données qui vont concrètement s'échanger sont des spécialisations de la donnée fédérée sur la plate-forme qui les héberge.
Du point de vue de la hiérarchie des traductions, le schéma de génération est le suivant :

[16] Cf. S.Abitboul, al.., *Data on the web*, Morgan Kaufmann, 2000.
[17] Cf. Normes ITU X.208 et 209, associées au langage SDL (recommandation Z 100 de l'ITU).

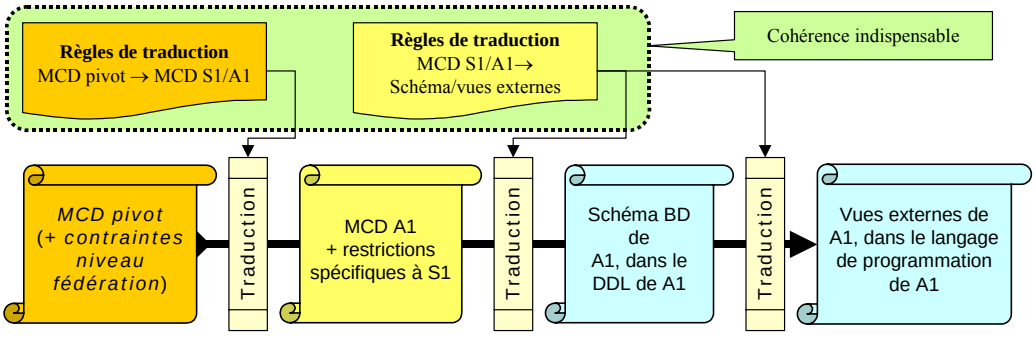


Figure 2.12 : Hiérarchie de traduction des modèles

Pour des raisons qui sont propres à l'existant des systèmes, le type d'une donnée particulière de la fédération peut être restreint, mais jamais l'inverse. De même, les contraintes au niveau de la fédération peuvent être renforcées au niveau d'un système, mais là encore jamais l'inverse. C'est la même relation qu'entre les TYPE et SUBTYPE du langage Ada, ou entre le schéma et les sous-schémas d'une base de données navigationnelle, ou entre vues dérivées et schéma relationnel. Le non respect de ces règles de typage va rendre non prédicatif les ensembles correspondant au type, ce qui engendrera des incohérences logiques et des ambiguïtés très difficiles à détecter lors de l'intégration[18].

Notons que les règles de traduction MCD S1/A1 → Schéma/Vues externes peuvent être non triviales comme, par exemple, lorsqu'il s'agit de dénormaliser des relations pour des raisons de performances (ce qui dans ce cas particulier implique de générer du code de transformation et des contrôles additionnels).

Lors d'un échange, matérialisé par un flux F1 entre des acteurs opérationnels des systèmes S1 et S2, les différentes transformations que subissent les données peuvent être organisées, conformément au schéma ci-dessous.

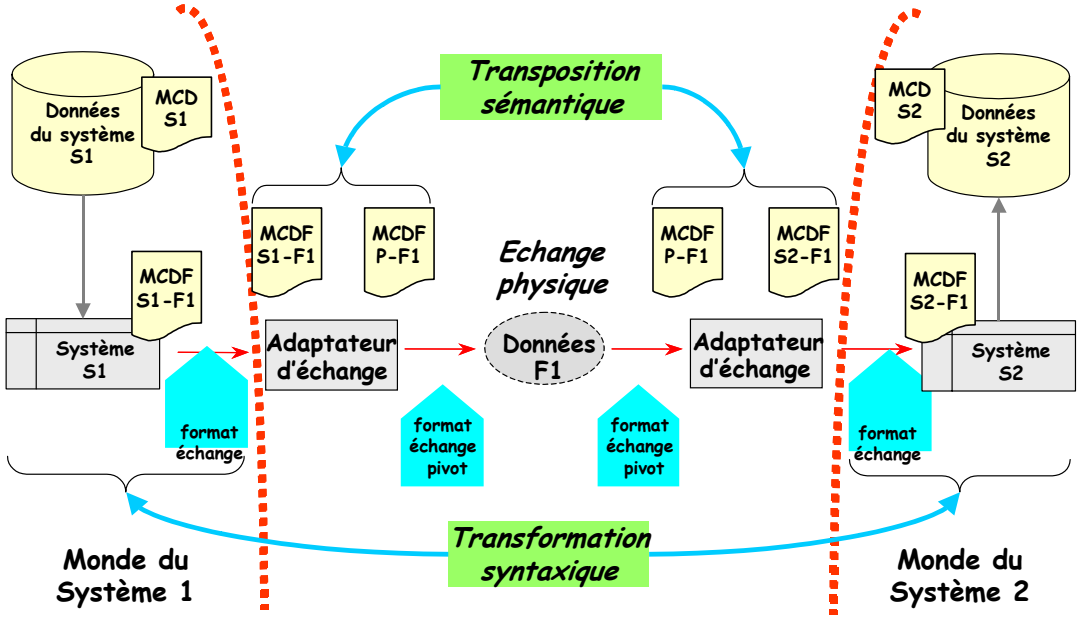


Figure 2.13 : Vision fonctionnelle d'un flux entre deux systèmes.

[18] Cf. J.Printz, *La programmation en univers incertain*, Revue Génie logiciel, N° 67, Décembre 2003.

On remarquera que les transformations opérées lors d'un échange sont les traductions inverses de celles de la figure 2.12.
En explicitant plus finement les règles de transformation on arrive finalement à des architectures de traitements tout à fait analogues à celles qui existent dans les compilateurs ou dans les SGBD.
On notera la similitude profonde entre ce schéma et la distinction introduite dans les années 70s entre syntaxe concrète et syntaxe abstraite dans les langages et les compilateurs[19] (voir chapitre 4, ci-dessous).
L'architecture fonctionnelle lors de l'émission et de la réception d'un message est strictement conforme au modèle de processus VEST (voir ci-dessous, chapitre 3). L'émission peut être explicitée comme suit :

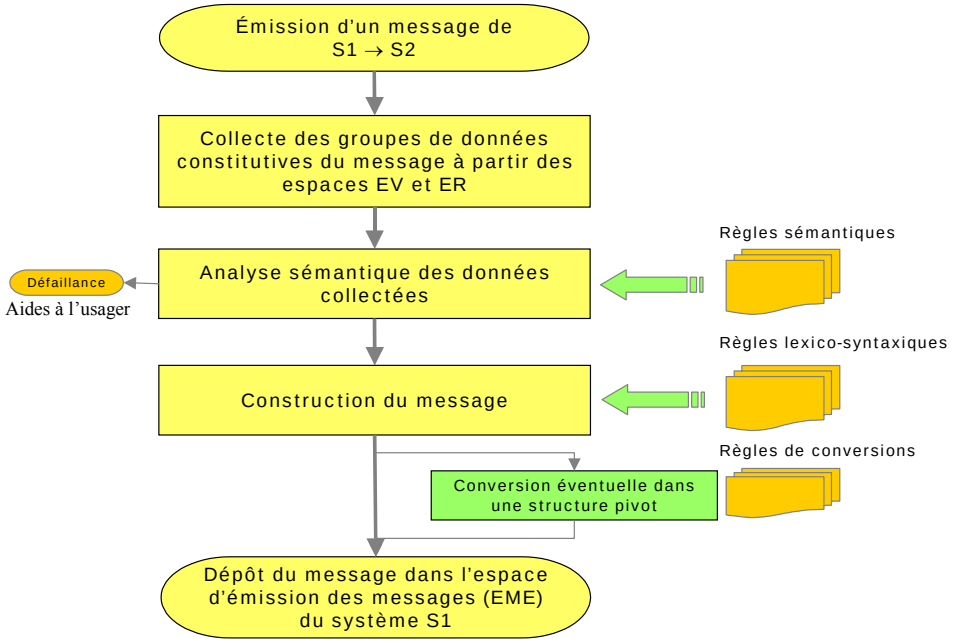


Figure 2.14 : Architecture de la traduction lors de l'émission d'un message.

L'architecture fonctionnelle à la réception du message est alors la suivante :

[19] En particulier à travers les travaux autour du langage PL1 fait par le laboratoire IBM de Vienne dans les années 70s.

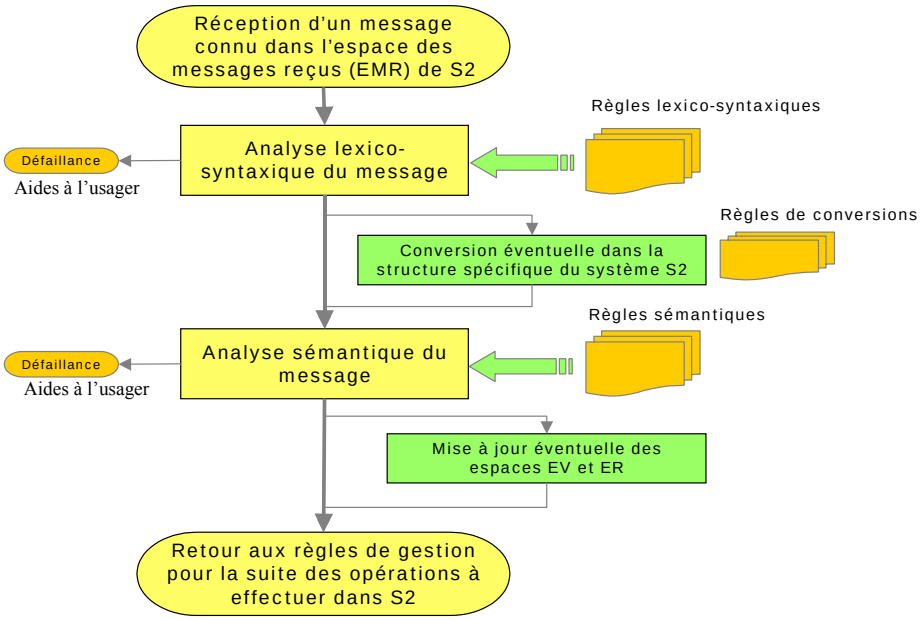


Figure 2.15 : Architecture de la traduction lors de la réception d'un message.

Les espaces EV et ER sont des espaces de travail pour les usagers de chacun des systèmes. L'espace de visualisation EV est un espace privé qui permet à un usager de valider visuellement le message ; rien n'empêche de mettre à disposition de l'usager différentes aides plus ou moins « intelligentes » pour faciliter le travail à ce niveau. L'espace de référence ER est un espace public qui ne contient, en théorie, que des données validées.
Les contrôles effectués lors des différentes étapes de l'analyse permettent de s'assurer que les messages émis et reçus sont toujours conformes aux règles de la fédération. L'application du modèle VEST, implique une certaine forme de redondance utile qui améliore la fiabilité globale de la chaîne de liaison.
NB : l'ensemble Réception – Emission réalise une transduction, qui intègre les différentes traductions nécessitées par le cheminement des données (Voir détails en Annexe 2 : *Notion de chemin de données*).

Le référentiel d'interopérabilité

C'est l'ensemble des modèles et des règles qui permettent de s'assurer que les données vont pouvoir s'échanger conformément aux règles établies pour la fédération.
A ce stade de l'analyse, ce référentiel contient :

- ✓ La nomenclature des types génériques de la fédération,
- ✓ Le MCD pivot et les contraintes qui lui sont propres,
- ✓ Les règles de traduction des types génériques et du MCD pivot dans les langages des différents MCD des systèmes (et les traducteurs correspondants) conformément au schéma de la figure 2.12.
- ✓ Les règles de codage et de décodage des messages associés aux flux d'échanges, conformément aux schémas des figures 2.14 et 2.15. L'implémentation de ces règles constitue un ensemble de services communs qui permettent de garantir que tout ce qui est émis et reçus par un système est codé et décodé conformément aux règles communes. Ceci est le cœur de l'architecture d'interopérabilité.

On remarquera que les modèles conceptuels de données pour les flux d'échanges (voir figure 2.13), ne préjugent pas des modèles conceptuels spécifiques à chacun des systèmes, ni de l'organisation des bases de données. Quelles que soient les décisions d'implémentation il est cependant impératif que toutes les traçabilités soient explicitées de façon à contrôler la propagation des dépendances fonctionnelles résultant des manipulations des données fédérées.

Intégration d'un nouveau système dans une fédération

L'intégration d'un nouveau système dans la fédération doit se faire conformément à la démarche ci-dessous :

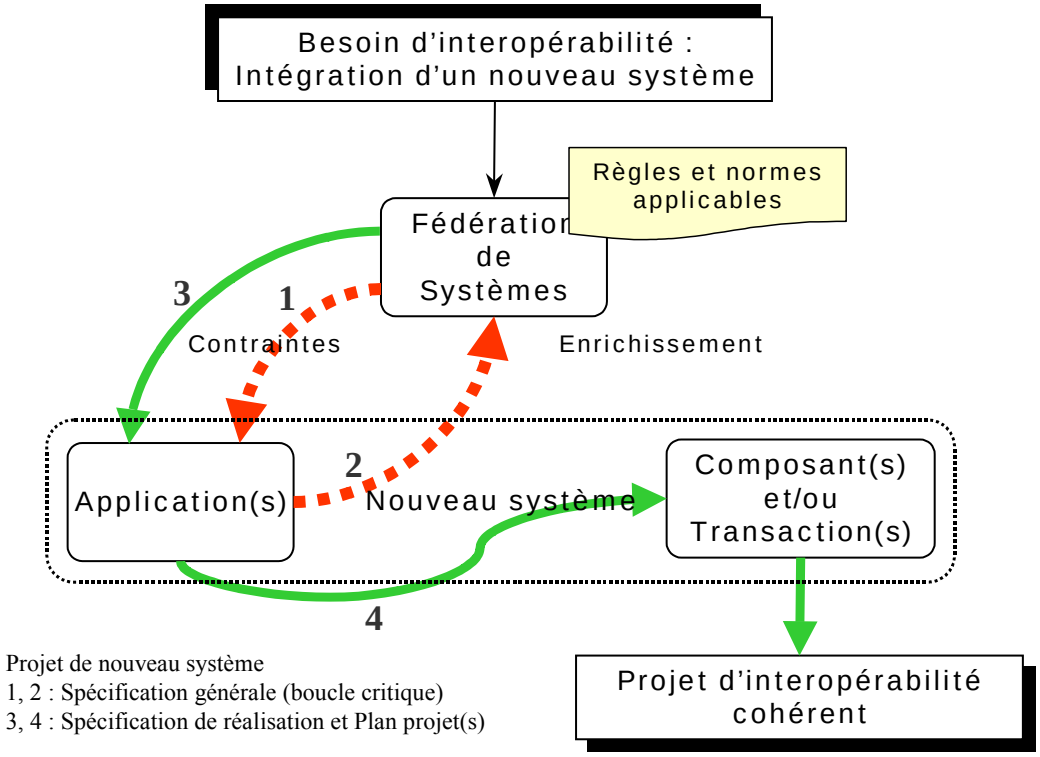


Figure 2.20 : Intégration d'un nouveau système dans une fédération

Le nouveau système hérite des données fédérées, dans la mesure où ces données ont un sens pour les opérationnels du nouveau système et propose à la fédération des données qui sont dans sa sphère de responsabilité, mais qui sont susceptibles d'intéresser d'autres systèmes de la fédération.
Dans ce dernier cas, il y aura évolution du référentiel d'interopérabilité. Les nouvelles données fédérées pourront être utilisées dans les anciens systèmes conformément au cycle de développement de chacun d'eux. Ceci exige que les données fédérées soient parfaitement gérées en configuration, selon le principe de la comptabilité ascendante que nous ne développerons pas ici.

3. LE BON USAGE DES MODELES

Il y a deux façons de considérer les modèles utilisés dans les projets informatiques.
Une première façon consiste à utiliser les modèles comme une discipline que l'on s'administre à soi-même afin de se mettre les idées bien en place et de s'efforcer

d'expliciter toutes les hypothèses. On recherche une formulation concise, de type idéographique, permettant une meilleure mémorisation et facilitant le raisonnement, un peu comme les figures que l'on utilisaient en géométrie afin de « raisonner juste sur des figures fausses ».

« Ce qui se conçoit bien s'énonce clairement » disait-on à l'époque de Port-Royal, et quoi de plus clair qu'un modèle. La physique a franchi une étape décisive, une vraie révolution[20], lorsque Galilée a érigé en principe que le langage de la science (la philosophie naturelle, comme on disait alors) ne pouvait être que le langage mathématique, le seul langage universellement adopté quelle que soit la langue et la culture des usagers des mathématiques. Newton, dans ses *Principia mathematica*, a utilisé le langage mathématique le plus élaboré de son époque, c'est à dire celui de la géométrie, pour exprimer sa compréhension du mouvement des corps, et non pas celui du calcul différentiel et intégral auquel nous sommes habitués. La mise à disposition d'un langage de modélisation unifié, raisonnablement partagé entre les acteurs qui ont affaire avec l'informatique, est indispensable à une bonne communication. Cependant, il faut rester modeste vis-à-vis de la réalité et des capacités de la notation ; chacun sait que les modèles mathématiques de la relativité générale sont inconciliables avec ceux de la mécanique quantique (qui elle même est encore objet de débat car une entité physique peut être à la fois, et simultanément, onde et corpuscule, sans être clairement localisée dans l'espace !) : la théorie unifiée reste un défi pour la physique mathématique, depuis maintenant plus de 80 ans ! Une bonne notation est nécessaire, mais pas suffisante : c'est ce que nous enseigne l'histoire des sciences depuis des siècles.

Les qualités d'un bon modèle sont connus, depuis longtemps : cohérence, clarté, complétude, économie et homogénéité des concepts, robustesse, etc. La raison d'être des modèles est de permettre au groupe qui les partage d'être plus efficace. La problématique étudiée à l'aide des modèles doit être ramenée à l'essentiel de façon à pouvoir raisonner sur une description des choses plus simple, en évitant l'écueil du simplisme où le modèle ne représenterait plus que des trivialités. Les règles de construction du modèle doivent être partagées, ou « évidentes » pour des gens partageant une même culture scientifique.

En informatique, toutes ces bonnes propriétés sont évidemment nécessaires, mais à l'usage elles s'avèrent insuffisantes, essentiellement pour cause d'évolutivité rapide des systèmes. Un système d'information est en perpétuelle évolution, et donc tous les modèles qui servent à le définir évoluent tout pareillement. Compte tenu du « turnover » naturel des personnels des projets, les modèles doivent être transmis d'un acteur à l'autre pour vivre et accompagner les acteurs dans leurs réflexions. De plus, la réalité sociale et économique auxquels les systèmes d'informations se confrontent est tellement diverse, pas nécessairement rationnelle (cf. la législation sociale, le code des impôts, la nomenclature du système de santé, etc. qui sont partie intégrante de nombreux systèmes d'information), qu'il est exclu que cette réalité puisse s'exprimer avec un seul langage, et que, plus encore qu'en physique, on sache tout modéliser[21].

D'où la règle d'ingénierie des modèles :

---

[20] C'est à ce type d'évolution profonde que s'applique la notion de changement de paradigme introduite par T.Kuhn, *Structure des révolutions scientifiques*.

[21] Voir les travaux de H.Simon, *Models of bounded rationality*, MIT Press, entre autres.

Règle d'obsolescence des modèles :
Tout modèle qui n'est pas utilisé informatiquement dans le procédé de construction et d'évolution de la fédération de systèmes deviendra inévitablement obsolète, pour cause de duplication et de redondance des informations.

C'est la raison fondamentale pour laquelle la seule référence qui fait foi, dans un système, est le code exécutable et/ou les textes qui permettent de générer automatiquement ce code exécutable ; d'où le soin qu'il faut donner à la spécification de ce socle. Tout ce qui n'est pas intégré dans le procédé de construction ne peut être considéré, au mieux, que comme un commentaire.
On peut considérer que plus il y a d'interactions entre les acteurs, et plus il y a d'acteurs avec des cultures disparates, plus vite apparaît l'obsolescence. Dans un projet d'interopérabilité, nous sommes au maximum de complexité, et donc au maximum de la rapidité de l'obsolescence, d'où la nécessité des traductions automatiques ***sans retouche***.
D'où **une seconde façon de considérer les modèles** utilisés dans les projets informatiques, et plus particulièrement dans les projets d'interopérabilité : ne seront considérés comme modèles, que les modèles réellement informatisés, c'est-à-dire des modèles dont la syntaxe et la sémantique sont suffisamment bien définies pour pouvoir être exploités automatiquement dans le procédé de construction du système à l'aide de traducteurs. Pour des informaticiens, le modèle le plus naturel et le plus fondamental est celui de machine[22]. C'est le concept de machine qui est le mieux à même de représenter de façon rigoureuse ce dont le modélisateur-concepteur de système a besoin, là où il en est dans son travail de modélisation. Seul le concept de machine est à même de donner du sens rigoureux à la nature fondamentalement transformatrice et comportementale du traitement de l'information.
Pour parler le langage de la phénoménologie, on peut dire que le concept de machine est l'essence même de la modélisation de l'information, l'apparaître, ou le « da-sein », de Husserl-Heidegger.
Dans la réalité informationnelle, tout se transforme, tout se négocie selon des priorités et des échéances temporelles qui elles-mêmes peuvent changer avec le temps ; tout est processus[23]. La modélisation de départ la plus naturelle doit donc être fondée sur la notion de processus.
Il y a de ce point de vue en parallèle tout à fait intéressant entre les procédés de construction et de transformation de l'information tel qu'on le voit dans l'ingénierie de l'information, et les procédés physico-chimiques qui sont ceux de la thermodynamique et de la physique statistiques qui mettent également au premier plan la notion de procédé[24].
Les langages de modélisation que nous pouvons utiliser doivent donc être jaugés par rapport à cette réalité fondamentale que sont les processus.

---

22 Dans la très vaste littérature sur le concept de machine, on peut citer : M.Minsky, *Computation, finite and infinite machines*, Prentice Hall ; Z.Mana, *Mathematical theory of computation*, Mc Graw Hill ; P.Denning & al., *Machines, languages, and computation*, Prentice Hall. Une recherche sur les mots clés ASM, ASML (abstract state machine language) est révélatrice.

23 Pour les amateurs de vraie philosophie, voir A.Whitehead (co-auteur, avec B.Russel des *Principia Mathematica*), *Process and reality*, Free Press ; ou encore M.Blondel, *L'action*, Le Seuil, sur un mode plus philosophique.

24 Cf. l'ouvrage de B.Ogunnaike, W. Ray, *Process dynamics modeling, and control*, Oxford University Press; souvent référencé dans la littérature « Agile », cf. notre ouvrage à paraître chez Hermès.

Dans les SIOC, la notion de processus utile à la modélisation métier[25], peut se schématiser comme suit :

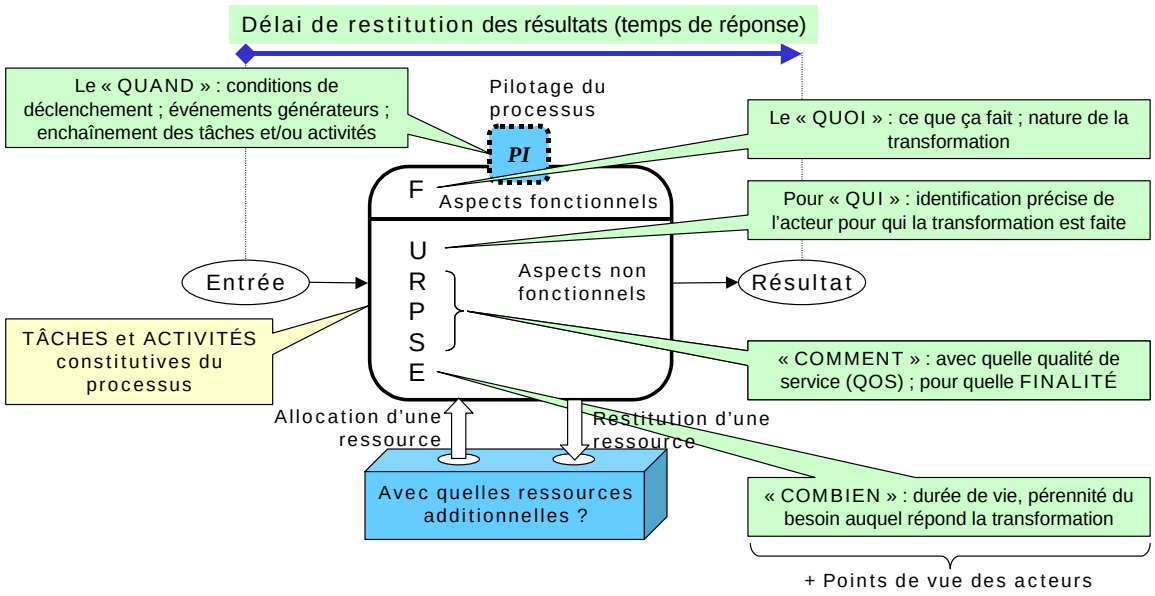


Figure 3.1 : Modèle générique de processus avec ses attributs métier.

Conformément à notre définition de la sémantique le processus est défini :

a) Par ce qu'il fait (aspects fonctionnels F), c'est-à-dire les transformations qui sont effectuées sur les entrées pour obtenir les résultats, et
b) Comment il le fait (aspects non fonctionnels, URPSE, qui sont les attributs qualité tels que définis dans la norme ISO/CEI 9126, pour Usability, Reliability, Performance, Serviceability, Evolutivity).

Tout ceci peut parfaitement s'exprimer en tant que besoin, au niveau des modèles métiers qui sont des modèles essentiellement qualitatifs. Associés à chacun de ces attributs on peut faire apparaître les acteurs qui ont à connaître l'attribut, de façon à bien faire ressortir les contraintes qui leur sont propres, lesquelles seront prise en compte ultérieurement dans les modèles de conception, puis dans le langage des plates-formes.
Par exemple, un acteur métier sera très sensible aux aspects facilité d'emploi (U) et fiabilité (R) pour des tâches /activités qu'il juge vitales pour la finalité de son travail, alors qu'un acteur chef de projet sera très sensible aux aspects risques en terme de coût, qualité, délai de la réalisation correspondante.
En fonction des tâches effectuées, le processus métier peut faire appel à des ressources additionnelles gérées par un autre processus, selon des modalités qui devront être explicitées. Cette situation est celle des processus de support dans les chaînes de valeur[26] des modèles d'entreprises. Une ressource allouée doit être restituée, sauf si il s'agit d'une ressource consommable ; dans ce cas c'est au processus support de gérer son stock pour ne jamais être en rupture.
Le pilotage PI réalise l'ordonnancement des travaux en fonction de ce qu'il sait être les priorités. C'est lui qui exécute le programme.

[25] Sur ce sujet, voir A.Scheer, *ARIS - Des processus de gestion au système intégré d'applications*, Springer France, 2002.
[26] Notion formalisée par M.Porter, dans ses deux ouvrages *Competitive advantage* et *Competitive strategy*, Free Press.

D'un point de vue plus informatique, mais qui reste cependant parfaitement fonctionnel et indépendamment de toute plate-forme informatique, le processus présuppose un modèle de calcul dans lequel on retrouve les constituants d'une machine générique apte à exécuter le processus.
Nous aménageons le schéma précédent comme suit (Modèle VEST) :

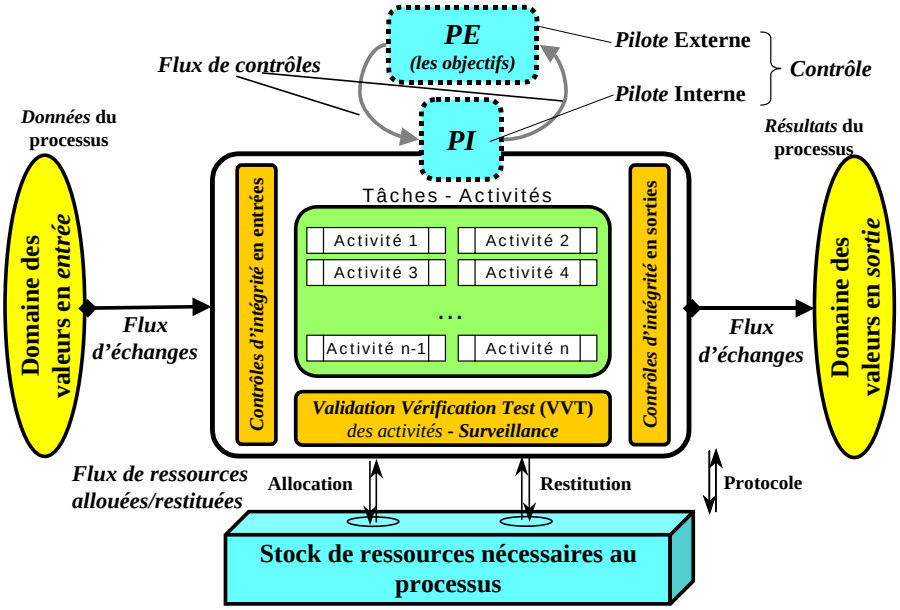


Figure 3.2 : Modèle générique de processus et fonctions d'intégrité VEST

Sur ce schéma, dont on remarquera l'orthogonalité, de nouvelles fonctions apparaissent. Par exemple, les fonctions de contrôle d'intégrité de ce qui entre et sort du processus ; il est inutile de démarrer les activités si les données et/ou informations en entrée sont erronées ou incomplètes.
La fonction SURVEILLANCE vérifie et valide tout ce que réalise les activités. Cette fonction est fondamentale car dans la réalité il y aura toujours des erreurs de la part des acteurs, donc des défaillances des activités et au final une défaillance du processus. La fonction SURVEILLANCE est le système immunitaire du processus ; elle joue donc un rôle fondamental dans la fiabilité du processus, et plus généralement dans la fiabilité des chaînes de liaisons qui s'exécutent au profit des acteurs opérationnels[27].
Les fonctions de contrôle d'intégrité des entrées et des sorties, ainsi que la fonction surveillance sont des éléments fondamentaux du système qualité attachés au processus, d'où le nom VEST – validation des entrées, des sorties, des tâches T – du modèle de processus.
Le pilote externe PE est l'autorité qui fixe au pilote interne PI les règles qu'il devra respecter dans l'ordonnancement des activités, ou les comportements à tenir en cas de défaillances. Dans tous les cas de figure, le pilote interne PI rend compte de ce qu'il fait au PE selon des modalités définies par le PE.
Dans des structures de pilotage de type matriciel, il peut y avoir plusieurs PE qui doivent agir de façon cohérente pour ne pas paralyser PI avec des ordres contradictoires. D'une façon plus générale, pour coller à la réalité, on ne peut pas

[27] Il est remarquable que cette notion, introduite dès l'origine des ordinateurs par Von Neumann dans ses études sur la fiabilité des automates, voir par exemple *Reliable computation in the presence of noise*, soit si peu connue ; aucune de nos machines ne fonctionnerait, si elle n'avait, bien cachée en leur sein, des circuits et des fonctions spécialisés dans la surveillance.

exclure que PE soit une structure de pilotage qui comporte plusieurs « centres » de décision[28]. Il est évident que la complexité sera beaucoup plus grande.

La correspondance métier – informatique.

Sans entrer dans les détails, il est clair que le modèle de processus ébauché ci-dessus est valide également pour les entités informatiques qui vont s'exécuter sur les plates-formes, et ceci jusqu'au niveau des instructions élémentaires de la plate-forme d'exécution (ce peut être une JVM, ou des machines de plus haut niveau). D'un point de vue de l'interopérabilité, il n'est évidemment jamais nécessaire de descendre aussi fin ; seuls vont nous intéresser les niveaux de granularité correspondant à ce qu'on appelle en informatique les applications et les transactions.
Le niveau de granularité de l'application est la base de données (une ou plusieurs) ; celui de la transaction est une transformation élémentaire sur un sous ensemble des données gérées par l'application.
La transaction est certainement l'une des notions les plus fondamentales de la mécanique informationnelle. Elle vaut tout à la fois au niveau métier où l'on n'a pas attendu l'informatique pour parler de transaction, et au niveau informatique où elle est apparue dans la mouvance des bases de données, lorsque le développement des réseaux a permis l'émergence du modèle de programmation transactionnel (OLTP/OLAP, pour « *On Line Transaction Processing* » ou « *On Line Analytical Processing* »).
Une transaction généralise, au niveau macroscopique métier, la notion d'instruction élémentaire indivisible au niveau de l'unité centrale d'un ordinateur. C'est un quantum d'action élémentaire et indivisible.
Le schéma logique d'une transaction est le suivant :

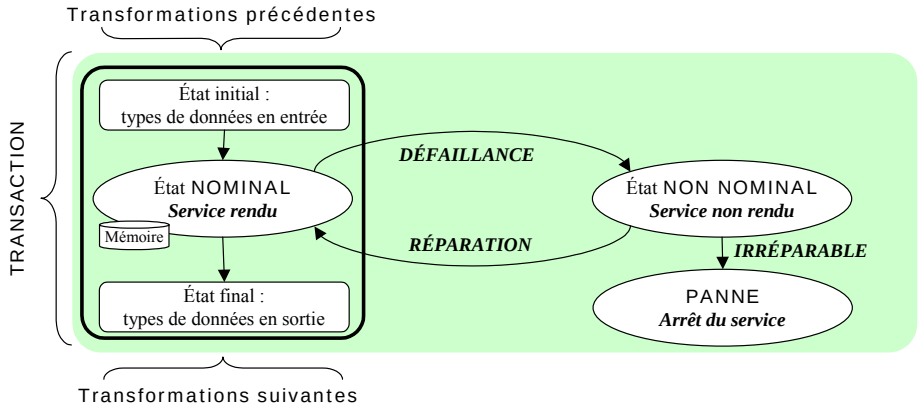


Figure 3.3 : La transaction comme quantum élémentaire de transformation

Pour que la transformation ait tous les attributs d'une transaction, il est nécessaire et suffisant que les propriétés dites ACID soient vérifiées. Ces propriétés définissent le modèle sémantique de la transaction.
Lorsque le moniteur d'activités lance une transaction, celle-ci ne peut se trouver, lorsqu'elle se termine, que dans deux états :

a) ou bien la transformation est faite, et alors elle est définitivement acquise (propriété de durabilité) ;

[28] A ce sujet, voir G.Allison, *Essence of decision*, qui est un grand classique, et T. de Montbrial, déjà cité.

b) ou bien les circonstances on fait que la transformation n'a pu être menée à bien, et alors tout ce qui a pu être fait de façon intermédiaire est purement et simplement annulé (propriété d'atomicité).

Lorsqu'une transaction est lancée, elle dispose de toutes les ressources dont elle a besoin pour s'exécuter, comme si elle était seule (propriété d'isolation). Enfin, lorsqu'elle se termine, la base de données doit être dans un état cohérent (c'est la propriété de consistance, i.e. « *consistancy* » en anglais, qui est un terme de logique mathématique[29]), c'est-à-dire que les invariants qui définissent la cohérence de la base de données sont tous VRAI.

Une transaction faite (on dit COMMITED) ne peut être défaite que par une transaction inverse explicite dite de compensation.

Par le biais des transactions, on introduit dans la mécanique informationnelle une notion de réversibilité, et donc de déterminisme, ce qui d'une certaine façon revient à remonter le temps. Cette propriété est fondamentale pour valider et vérifier les applications distribuées[30]. Quand bien même elles s'exécutent en concurrence, tout se passe comme si elles avaient été sérialisées, ce qui constitue la propriété d'isolation.

Le modèle de programmation transactionnelle intègre un modèle de panne qui nous permet de spécifier quoi faire en cas d'incident, indispensable pour la surveillance et l'intégrité. On notera que la logique sous-jacente à ce modèle est tri-valente[31] : {FAIT, NON FAIT REPARABLE, NON FAIT NON REPARABLES}, et qu'en conséquence la règle logique du tiers exclus ne marche plus.

Une application n'est qu'un regroupement de transactions (que l'on peut organiser en services), ce qui permet d'allouer un pool de ressources dans lesquelles les transactions vont aller puiser ce dont elles ont besoin pour s'exécuter. Les critères de regroupement sont très divers : métiers, techniques, organisationnels, non fonctionnels, etc. D'où la difficulté d'obtenir des partitions sans recouvrement.

L'enchaînement des transactions résulte des interactions de l'application avec son environnement. L'enchaînement de une ou plusieurs transactions réalise un service. Si le service est lui-même une transaction, on a alors affaire a une transaction dite « longue », par opposition aux transaction précédentes qui sont « courtes ». Dans le monde réelle la plupart des transactions sont longues ; le modèle d'intégrité est plus complexe, nous n'en parlerons pas ici.

Ce qui est vraiment fondamental du point de vue de la modélisation est la traçabilité de la correspondance entre le modèle métier et le modèle informatique en terme d'applications et transactions.

C'est ce que montre le schéma suivant :

29 Cf. A.Tarski, *Introduction à la logique*, Gauthier-Villars et S.Kleene, *Introduction to metamathematics*, North-Holland..

30 Cf. R.Orfali & al., *Client/server survival guide*, Wiley, 1999.

31 Dans la réalité, c'est une logique modale correspondant au différents états de retour possibles ; à ce sujet, voir, J-Y.Girard, *La logique linéaire*, dans Pour la science, N°150, Avril 90 pour une introduction.

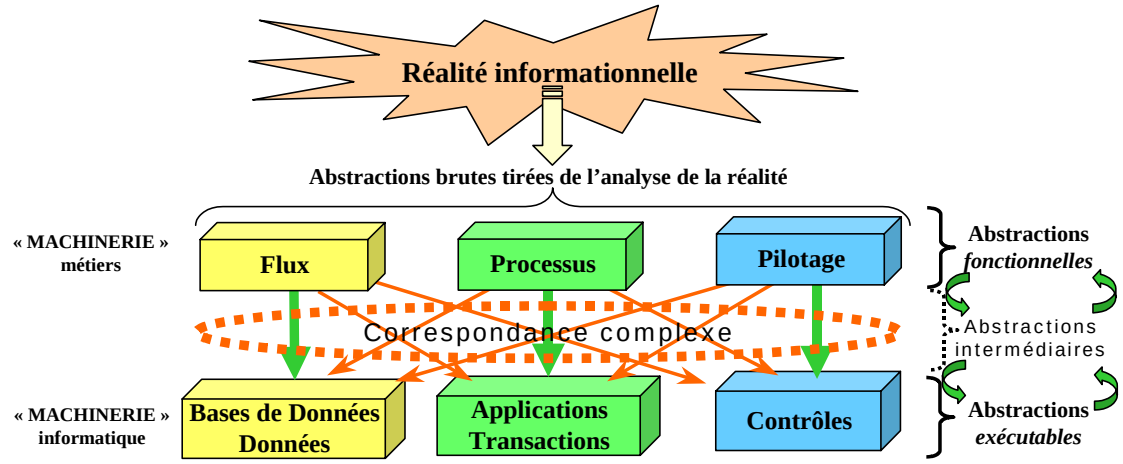


Figure 3.4 : Correspondance métier ↔ informatique.

Cette correspondance est intrinsèquement complexe car, certes il y a bien une correspondance naturelle FLUX ↔ BASE DE DONNEES, PROCESSUS ↔ APPLICATIONS-TRANSACTIONS, PILOTAGE ↔ CONTROLES, mais cette correspondance n'épuise pas les combinaisons possibles.

Par exemple un flux va bien sûr générer des parties de modèle de données, mais il va également générer des fonctions de conformité des données portées par le flux, qui elles-mêmes vont générer des actions de contrôle.

Si l'on rajoute à ce schéma la dimension interopérabilité, tous les organes se dédoublent (voir ci-dessous, chapitre 4), ce qui fait que au lieu d'un schéma simple à 3 + 6 flèches on obtient un schéma à 6 + 30 flèches (pour la correspondance métier ↔ informatique), auquel il faut rajouter les nouvelles correspondances issues du dédoublement du plan informatique, soit 18 flèches de plus, pour gérer les dépendances fonctionnelles potentielles entre les éléments fédérés et les éléments spécifiques du système considéré.

En terme de combinatoire statique, les combinaisons deux à deux de N entités varient comme $\frac{N(N-1)}{2}$ soit une complexité statique en $O(N^2)$; alors qu'en combinatoire dynamique où il faut intégrer tous les cheminements possibles, il faudra considérer l'ensemble des parties, soit 2^N.

L'objectif fondamental de la modélisation est donc de contrôler la taille de l'ensemble des correspondances possibles, de façon à ce que l'ingénierie correspondante reste dans le domaine du raisonnable, c'est-à-dire linéaire, ou localement en N^2. Toute combinatoire supérieure sera généralement fatale au projet, d'ou une règle de bon sens de l'ingénierie de l'interopérabilité.

NB : Les correspondances complexes qui relient les entités métier aux entités informatiques (transactions et/ou données) ne sont pas des arborescences mais probablement des treillis, voire même des catégories. D'où les grandes difficultés à les faire évoluer de façon cohérente.

Règles d'ingénierie :
Ne jamais rendre interopérable ce qui n'a pas lieu de l'être.
Appliquer systématiquement et intelligemment la règle de subsidiarité (c'est à dire faire baisser la complexité, sinon la subsidiarité ne sert à rien) quitte à développer quelques traducteurs supplémentaires.
Adopter des règles de partition des ensembles de données qui intègrent la dimension dynamique des traitements de façon à rendre explicite les dépendances fonctionnelles entre les données.
Dans tous les cas de figure, automatiser ce qui peut l'être (une combinatoire en N^2, si N n'est pas trop grand, reste gérable avec la puissance des machines dont nous disposons aujourd'hui ; une combinatoire supérieure est ingérable et mettra le projet en grand péril).

Machine informationnelle

La machine informationnelle est fondée sur le concept de machine abstraite, indépendamment de la plate-forme informatique qui interprètera la machine abstraite le moment venu (dans le langage MDA, cette machine serait un PIM, *Platform Independent Model*).
Le concept de machine abstraite sera repris ci-dessous. L'important est de signaler que ces machines abstraites vont permette de spécifier des plates-formes avec une sémantique explicite sur laquelle on pourra construire les applications et les systèmes d'information répondant aux besoins.
Les plates-formes ébauchées dans les schémas qui suivent sont conformes au pattern *Model View Controler* (MVC) dont l'usage est relativement généralisé pour les applications clients-serveurs.

Modèle de plate-forme générique basée sur le pattern MVC

Le premier niveau de description de la machine informationnelle fait apparaître un univers de serveur constitué de trois types principaux :

a) Le serveur d'IHM. Il prend en compte la facilité d'emploi requise pour telle ou telle catégorie d'acteurs et les différents rôles qui leur sont attribués.
b) Le serveur d'applications. Il prend en compte la logique métier, ce qui peut conduire à considérés différents serveurs spécialisés sur un thème métier particulier.
c) Le serveur de données. Là encore, on peut les organiser en fonction de la nature des données traitées. Un serveur de données peut gérer une ou plusieurs bases de données.

Ces différents serveurs[32] constituent autant de machines abstraites dont on parlera au chapitre 4.
Une machine informationnelle contient toutes les ressources nécessaires à l'accomplissement de la mission de pilotage que les opérationnels effectuent au profit de telle ou telle unité active.
Une machine informationnelle peut héberger un ou plusieurs systèmes d'information. C'est une entité homogène d'administration et d'allocation de ressource qui peut être

[32] Sur ces notions de serveurs, avec une orientation plate-forme matérielle, voir R.Chevance, *Serveurs multiprocesseurs, clusters et architectures parallèles*, Eyrolles, 2000, qui est une mine d'informations rassemblées par quelqu'un qui connaît son sujet.

placée sous la responsabilité d'un décideur qui arbitrera les allocations de ressources en fonction de la situation de saturation des systèmes et des nécessités résultant de l'environnement des systèmes (pertes de ressources, reconfiguration, etc.).
La machine administre toutes les ressources qui lui ont été attribuées via un gestionnaire de configuration dans lequel toutes les ressources disponibles sont décrites.

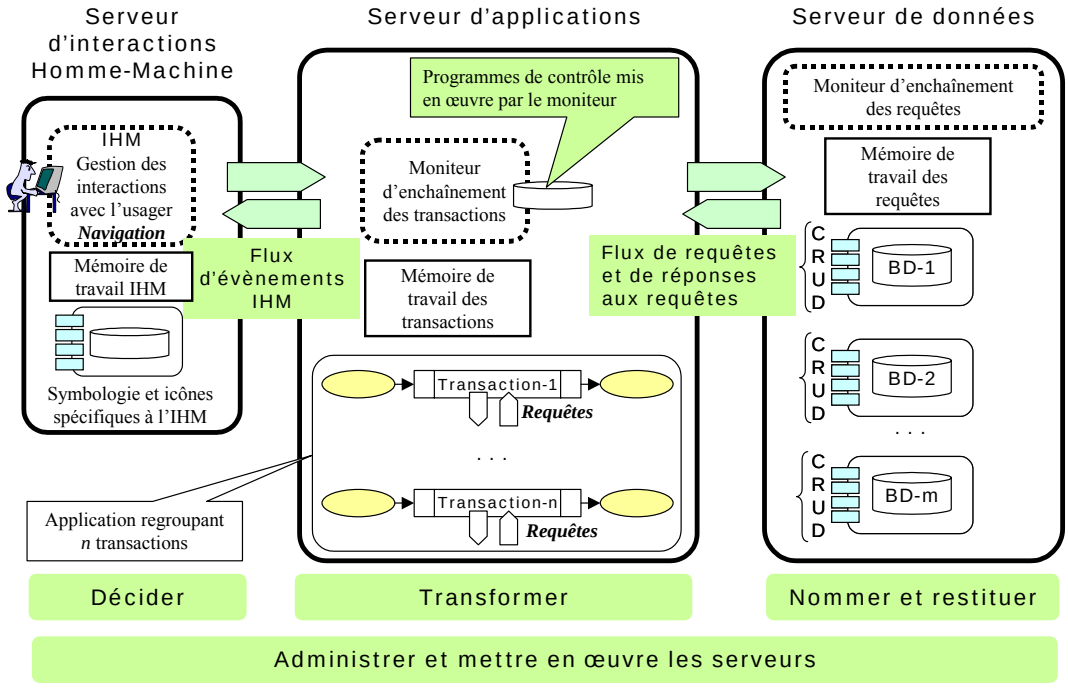


Figure 3.5 : Modèle de machine MVC générique

En situation d'interopérabilité, ce niveau de modélisation, bien que fonctionnellement complet, est insuffisant car en aucun cas la ressource communication ne peut être considérée comme banalisée ; elle doit être explicitée. Utiliser les services de La Poste, de DHL, ou la valise diplomatique est fondamentalement différent du point de vue des caractéristiques non fonctionnelles.
La première raison est l'impératif d'autonomie et la capacité de survie des UA et des SIOC qui les pilotent.
La seconde raison est de nature plus technologique (radio, filaire, fibre optique, etc.), avec la prise en compte de caractéristiques qualité dimensionnantes (sécurité, fiabilité, performance, etc.).
D'un point de vue administration, il est indispensable de faire apparaître, a minima, deux niveaux d'administration :

a) Une administration de ressources attribuées en propre à une machine informationnelle MI. Cette MI peut héberger un ou plusieurs SIOC, mais certainement pas tous les SIOC présents sur un théâtre. La MI administre ses ressources en optimisant sa capacité de survie, comme pourrait le faire un moniteur de machines virtuelles dans les systèmes d'exploitation qui possède cette fonctionnalité[33].
b) Une administration de ressources mise en commun, au profit des différentes MI qui interopèrent. Les ressources correspondantes sont gérées de façon ad hoc, à

[33] Assez rare, et toujours en solution propriétaire ; voir par exemple chez IBM l'offre « business on demand » ex VM-CMS.

la demande, avec comme critère d'optimisation le maintien de la capacité d'interopérabilité des différentes MI.

Dans le premier cas on peut s'appuyer sur des mécanismes du type CORBA, OLE/DCOM, ou des plates-formes complètes comme J2EE, Web-sphere/Eclipse ou .NET selon les adhérences admises à tel ou tel fournisseur.

Dans le second cas, il faut une infrastructure d'échange gérée comme telle, comme on en trouve dans le réseau de télé-compensation des banques, ou celui des billetteries des compagnies aériennes, etc. Une MI qui souhaitent interopérer se connecte à un point d'accès au réseau commun, selon les modalités techniques de raccordement prévues.

Le schéma logique de cette articulation est le suivant :

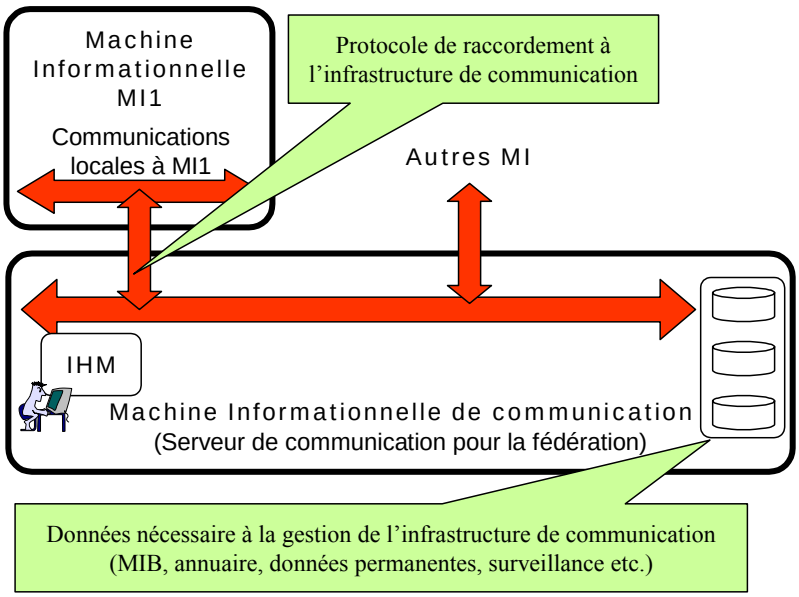


Figure 3.5a : MVC générique et son serveur de communication pour services distants

L'infrastructure d'échanges entre les systèmes fédérés peut être vue comme une machine informationnelle à part entière, bien que spécialisée sur un domaine technologique lui-même très complexe, dont la mise en œuvre peut elle-même nécessiter plusieurs MA spécialisées.

Prise en compte des caractéristiques de la ressource communication

Il faut distinguer les serveurs locaux et les serveurs distants car les sémantiques d'appels sont évidemment très différentes, comme indiquées dans le schéma ci-dessous :

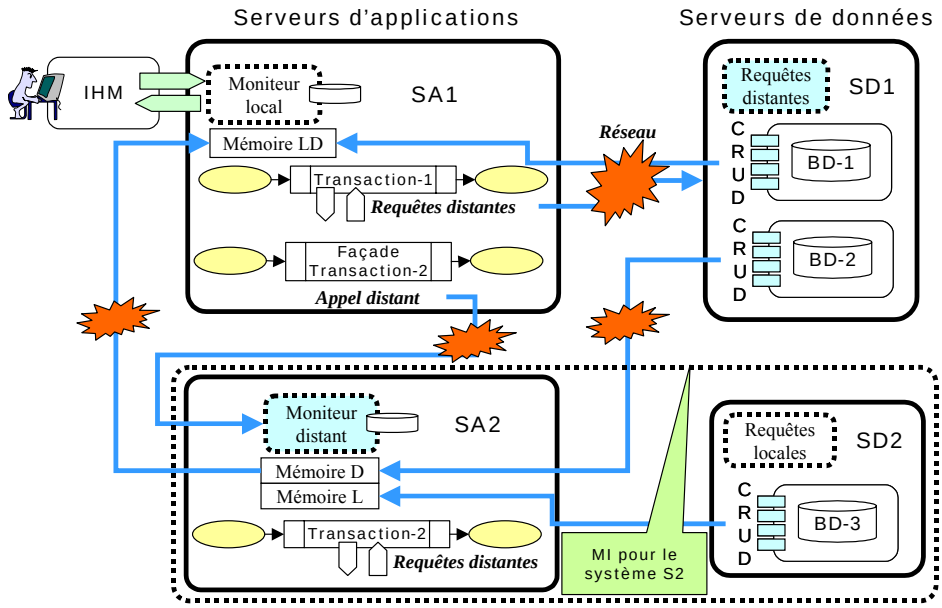


Figure 3.6 : Modèle de machine MVC généralisée – Actions locales et distantes

NB : les → indiquent quelques uns des chemins de données utilisés depuis les serveurs de données locaux et/ou distants, jusqu'au poste de travail de l'usager. Ceci est essentiel pour la traçabilité des différentes traductions effectuées tout au long de ces chemins.
En toute logique, aucune fonction critique de SA1 ne doit être accédée de façon distante car on ne peut jamais être sûr d'une disponibilité de 100% du réseau de communication.
La répartition des ressources doit être faite selon les critères de disponibilité des différentes applications des différents systèmes.
Dans tous les cas de figure, il faut une gestion de configuration et une administration nécessairement complexes[34], compte tenu de la nature des services demandés aux systèmes.

Structure d'un projet d'interopérabilité – gestion de l'évolutivité.

Un projet d'interopérabilité est complexe par nature car il consiste à coordonner plusieurs programmes (au sens DGA[35]) dont chacun peut nécessiter l'action conjointe de très nombreux acteurs. Un projet d'interopérabilité va mettre en jeux toutes les techniques de l'ingénierie système[36].
La structure générale d'un tel projet peut être schématisée comme suit :

[34] Voir l'approche *Autonomic Computig*, préconisée par IBM.

[35] Cf. Cavaillès, *Méthode de management des programmes d'armement*, Teknéa.

[36] Cf. JP.Meinadier, *Le métier de l'intégration système*, Hermès.

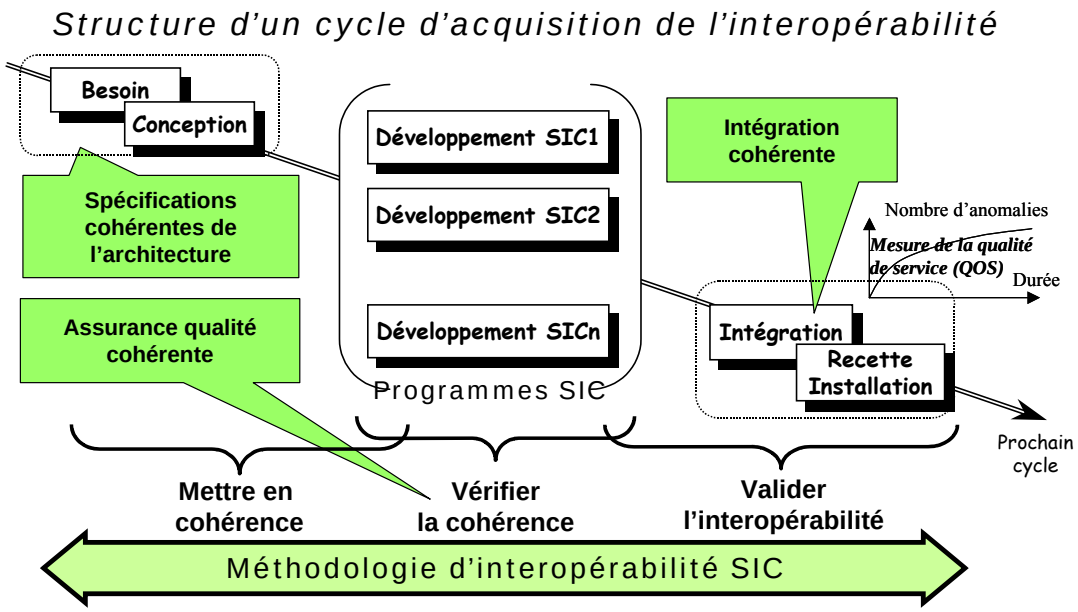


Figure 3.7 : Conduite d'un projet d'interopérabilité

La spécification cohérente de l'architecture est la définition rigoureuse, à l'aide de modèles, de la mécanique d'échange entre tous les systèmes qui ont décidé d'interopérer.
Cette spécification est un standard au sens le plus fort du terme. Son respect absolu est une condition nécessaire à l'obtention de l'interopérabilité qui sera vérifiée et validée lors de la phase d'intégration du système, sur un niveau de cohérence donné.
Pour satisfaire le besoin d'évolutivité des différents systèmes, il est indispensable que le standard d'interopérabilité évolue de façon cohérente, ce qui fait qu'à un certain instant t, on peut se trouver avec des systèmes qui sont à des états différents de standardisation.
La prise en compte de l'évolution des normes est fondamentale.
La fédération des systèmes doit supporter différents niveaux de normes. A l'instant t, plusieurs niveaux de normes peuvent donc cohabiter.
A l'instant t, un système travaille avec un niveau de norme particulier. Il émet et reçoit des informations qui doivent être compatibles ou comprises par ce niveau.
Le schéma de principe est le suivant.

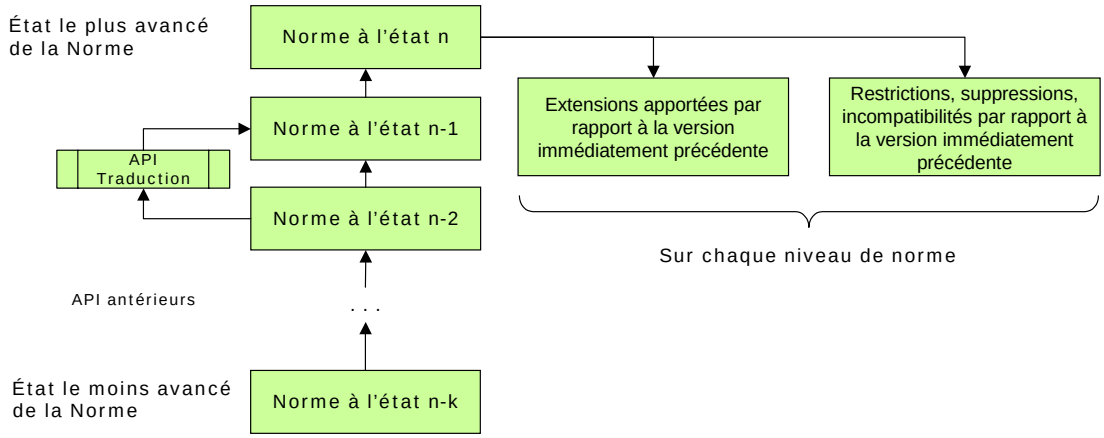


Figure 3.8 : Gestion des niveaux de normes

Chaque niveau doit pouvoir être traduit dans le niveau qui lui est immédiatement supérieur, aux restrictions, suppressions et incompatibilités près.
Au moment où l'on normalise le niveau *n*, il faut définir le traducteur qui permettra la migration du niveau $n-1$ vers le niveau *n*, et éventuellement le traducteur inverse.
A l'instant t, on dispose d'un jeu de traducteurs qui permet de ramener le niveau le moins avancé vers le plus avancé, avec quelques restrictions éventuelles.
Le mécanisme est le même en intra-système et en inter-système.

Approche fonctionnelle permettant de traiter simultanément différentes version de normes

A l'instant t, un récepteur reçoit une information d'un système qui se trouve à un certain niveau de norme. Trois cas sont possibles :

1. Cas 1 : les systèmes sont au même niveau. Il n'y a pas de problème.
2. Cas 2 : Le récepteur est à un **niveau supérieur** à celui de l'émetteur.
3. Cas 3 : Le récepteur est à un **niveau inférieur** à celui de l'émetteur.

Fonctionnement pour le cas 2

Le récepteur connaît les états antérieurs de la norme et dispose du jeu complet des API traducteurs.
Deux éventualités sont possibles :

1. On sait traduire l'ancien dans le nouveau, par simple application des traducteurs. Pas de problème, sinon de performance.
2. La traduction signale des erreurs dues aux restrictions, suppressions et incompatibilités. Une intervention de l'opérateur est possible pour éventuellement compléter l'information.

Fonctionnement pour le cas 3

C'est le cas le plus difficile car par définition le récepteur ne connaît pas les nouveaux états des normes.
Plusieurs éventualités sont possibles selon la façon dont la norme a été construite.
Le message est rejeté en bloc par le récepteur. Celui-ci signale à l'émetteur le niveau de norme qu'il est capable de comprendre. C'est à l'émetteur de s'adapter au niveau de norme du récepteur et de re-émettre le message. La disponibilité de l'adaptation doit être limitée dans le temps.
La norme est organisée en articles qui peuvent être à des niveaux différents. Ceci ouvre la possibilité que, bien qu'en étant à des niveaux globalement incompatibles, certains articles qui n'ont pas bougé peuvent cependant être compris (NB : Les normes des langages de programmation ont été construites selon ce principe ; si l'on n'utilise pas les nouveaux traits de langage, on peut tout de même compiler avec les anciens compilateurs tout en étant compatible avec les nouveaux programmes). L'information correspondante doit signaler le niveau global de la norme et le niveau de l'article.
La structure de la norme, et la façon dont la norme est implémentée dans les systèmes est très importante. Une norme bien structurée permettra tout de même de traduire, même si ce n'est que partiellement.
Les API de traduction et les traducteurs associés font partie des standards de la fédération.

4. ARCHITECTURE ET LANGAGE PIVOT

Eléments d'architecture pour l'interopérabilité

Lorsqu'on parcourt la littérature MDA, on est pris de vertige devant le déballage terminologique et la logorrhée verbale, un vrai méta-délire, dont on se demande à quoi et à qui tout cela va servir, et pour quel méta-client. Comment en est-on arrivé à ce degré zéro de la communication qui consiste à rendre systématiquement obscur et abscons des notions élémentaires et basiques, connues depuis des décennies. Cela repose, une nouvelle fois, la question du contenu des formations à l'informatique et de ce qu'il est impératif de connaître pour manipuler avec un minimum de risque la « matière informationnelle » concrète des projets informatiques, bref, de se comporter en ingénieur et non en bricoleur, fut-il très astucieux[37].

Langage et Méta-langage

La distinction langage méta-langage est devenue classique consécutivement aux travaux de G.Frege et des logiciens qui ont travaillé à la clarification des paradoxes de la théorie des ensembles, au tout début du 20ème siècle. On peut trouver une des formes les plus fouillées de ces réflexions dans l'ouvrage de R.Carnap, *The logical structure of language*, édité en 1935. N.Chomsky, linguiste éminent, donnait en 1959 un texte fondateur consacré aux grammaires formelles[38]. Plus près de nous, un ouvrage comme : *Notions sur les grammaires formelles*, de Gross et Lentin, publié en 1967, rendait cette problématique accessible aux informaticiens. Depuis, une abondante littérature concernant les compilateurs, et plus généralement les traducteurs, a vu le jour avec souvent des textes de très grande qualité qu'il est impératif de connaître si l'on veut maîtriser l'approche MDA, ou simplement comprendre ce que signifie paramétrer un programme.

Ce que les logiciens nous ont enseigné est que, pour parler de façon pertinente d'un langage, il faut un autre langage appelé meta-langage. Confondre les deux conduit directement à des paradoxes. Dire que : « Paris a 5 lettres » est une propriété méta linguistique, car on donne un information sur la façon d'écrire le mot « Paris » qui n'a rien à voir avec le sens de « Paris, capitale de la France ». Un exemple d'erreur liée à la confusion des niveaux consisterait à dire, par exemple : « 343 a 3 chiffres, or $343 = 7^3$, donc 7^3 a 3 chiffres ». Dans les aphorismes du chat, de Ph.Gelluck, on trouve des formulations paradoxales du type « c'est dingue, le mot long est plus court que le mot court », de structure méta-linguistique analogue.

Deux niveaux sont nécessaires et suffisants pour ne jamais se prendre les pieds dans le tapis. Par rapport à une problématique donnée, ce qui est difficile est de classer ce qui est linguistique, spécifique à la problématique, et ce qui est méta-linguistique, c'est-à-dire ce qui sert à la décrire ; c'est la distinction données / méta-données. En fait, cette frontière n'est jamais nette, ni dans les langages de programmation, ni a fortiori dans les langages naturels : la distinction est souvent de nature purement conventionnelle.

Dans une langue, le dictionnaire et la grammaire sont des entités méta-linguistiques.

La grammaire permet de générer des phrases grammaticales correctes (i.e. VRAI du point de vue de la grammaire) ; quant à la vérité sémantique, c'est autre chose.

---

[37] Voir le post-scriptum.

[38] Cf. *On certain formal properties of grammars*, Information and control, Vol 2, N°2, Juin 1959.

Dans la réalité, les modes d'expressions sont plus subtils et plus variés. Dans certains milieux, l'usage est d'utiliser une durée pour dénoter une distance, comme dans le message : « A 10 km de l'arrivée, les échappés ont 1 min 30 d'avance », à la limite de l'ambiguïté ; en montagne on indique les distance en heures de marche, ce qui permet d'intégrer le dénivelé ou la difficulté du terrain ; un responsable d'unité active dira qu'il dispose de 3 jours de vivre, ce que le logisticien traduira en tonnage, cubage, etc.
Chomsky avait introduit une notion de phrase a-grammaticale mais cependant sémantiquement compréhensible : « vous faire moi rigoler » est incorrect, mais compréhensible, surtout si c'est un étranger qui la prononce.
Les notations qui en résultent arrivent dans le monde de l'informatique avec la spécification du langage Algol 60, et l'apport décisif de J.Backus et P.Naur, de ce qui deviendra la notation BNF en 1958-1959[39].
Cette distinction entre la construction d'un programme avec les règles du langage, et la construction du langage avec les règles, grammaticales exprimées en BNF est certainement l'une des choses les plus importantes à bien comprendre si l'on ne veut pas modéliser des idioties ou des banalités.
Toute confusion entre les deux univers créera immédiatement d'énormes difficultés d'évolution de l'un et de l'autre, compte tenu de l'intrication du meta-langage dans le langage.
Le terme d'intrication avait été introduit par J-P.Desclée[40] dans sa tentative de modélisation d'un noyau grammatical commun à toutes les langues naturelles pour montrer que le mélange observé dans une phrase est plus que de l'imbrication. C'est un des aspects du langage naturel qui rend la traduction automatique virtuellement impossible.
Vouloir représenter la langue et la méta-langue avec la même notation est un exercice de style totalement dénué d'intérêt pour le modélisateur qui s'efforce tout au contraire de bien distinguer les deux. C'est mettre la confusion au cœur du dispositif de modélisation en faisant croire de surcroît que savoir modéliser, c'est savoir jongler avec une notation. Montrer que tout langage est exprimable dans le λ-calcul, y compris le λ-calcul lui-même, c'est « démontrer » l'universalité de la calculabilité. Pour un praticien, cela veut dire que des langages correctement construits peuvent être traduits les uns dans les autres (c'est une propriété fondamentale), mais il serait absurde d'un point de vue pragmatique, et surtout pédagogique, d'en déduire que tout doit être écrit en λ-calcul, et qu'en apprenant le λ-calcul on saura tout faire. Le λ-calcul constitue un langage pivot de tout ce qui est calculable[41] ; étant très général, il est sémantiquement très pauvre.
Lorsqu'on invente des formalismes, une bonne discipline intellectuelle consiste à les décrire dans la notation BNF. Bien souvent, on n'y arrive pas. Ou alors, il y a autant de règles que d'entités ! Ce qui veut simplement dire que l'on n'a pas identifié la frontière syntaxe-sémantique et que la compréhension de la problématique est insuffisante pour modéliser ce qui est réellement important. Manier correctement les formalismes[42] est intrinsèquement difficile : restons modestes et sobres.

---

39 Cf. R.Wexelblat, *History of programming languages*, ACM monograph series. Il existe une norme ISO/CEI 14977, Syntactic metalanguage – Extended BNF.

40 Voir en particulier les travaux du laboratoire de linguistique formelle, à Paris VII, vers la fin des années 70s, sur le programme PITFALL. En anglais, le terme est « entangled » que l'on peut également traduire par enchevêtrement. En mécanique quantique, on parle également d'histoire et d'états enchevêtrés, cf. R.Omnès, *Comprendre la mécanique quantique*.

41 Voir l'avant-propos de J-Y.Girard, dans *La machine de Turing*, Point science, Le Seuil.

42 Cf. L.Ladrière, *Les limitations internes des formalismes*, re-édition J.Gabay.

En application de la bonne vieille règle du rasoir d'Occam, ne créons pas de catégories inutiles. Essayons simplement de bien raisonner sur les deux niveaux, langage et méta-langage, c'est déjà assez difficile comme cela. D'un point de vue pratique, privilégions les systèmes de notations qui distinguent clairement les deux niveaux.

Langages et Machines

La notion de langage est indissociable de la notion de machine. Le formalisme des *Principia Mathematica*[43] est exprimable dans celui des machines de Turing, et inversement. Le fait est connu, et précède l'invention des premiers ordinateurs. Ce sont deux points de vue d'une même réalité profonde. Gödel et Turing ont démontré le même théorème en utilisant des symbolismes radicalement différents.
Certaines choses s'expriment mieux dans un formalisme fonctionnel, ou un système de re-écriture, et d'autres, de nature plus discrète et évènementielle, induisent immédiatement la notion d'automate, et conséquemment celle de machine.
Les architectes de compilateurs ont su utiliser pleinement cette dualité[44] dans les années 70-80, en s'appuyant, tantôt sur le formalisme des grammaires pour produire et générer du texte, tantôt sur le formalisme des automates pour reconnaître et piloter la traduction, selon les besoins, dans telle ou telle étape de la compilation. Ceci a permis de développer des architectures de compilateur, hautement paramétrables appelées « compiler-compiler » ceci dès la fin des années 70s.
L'approche MDA aurait certainement tout intérêt à se ressourcer auprès de ces technologies, comme d'ailleurs de la technologie des SGBD, afin d'éviter de réinventer la roue, et de semer la confusion, là où on devrait tout faire pour l'éviter[45]. Raisonner avec des machines abstraites est un moyen simple d'exprimer une sémantique que l'on saura partager avec de nombreux acteurs et manipuler sur des ordinateurs réels ; ce que l'on ne peut évidemment pas faire avec la sémantique dénotationnelle ou les domaines sémantiques de D.Scott[46].
Lorsqu'on spécifie une sémantique, une bonne discipline d'ingénieur est d'abord d'essayer de la définir à l'aide d'une machine abstraite[47], en isolant parfaitement la partie instructions (i.e. le jeu d'instructions et ses invariants), la partie contrôle des enchaînements (i.e. les transitions), la partie états des opérations effectuées (i.e. les évènements). Si on n'y arrive pas, c'est que la sémantique n'est pas décantée, et que, en conséquence, il est illusoire de vouloir construire le langage de la machine qui sera nécessairement mal ficelé.
La structuration d'un jeu d'instructions reste un exercice d'une grande complexité qui requiert une connaissance approfondie des capacités de hardware et une compréhension profonde des techniques de compilation, comme l'ont montré les travaux sur les architectures RISC, dans les années 80. Il ne faut pas s'attendre à ce que l'exercice soit plus simple lorsque l'on veut monter le niveau d'abstraction des plates-formes d'exécution vers des entités métiers. A tout le moins, il est nécessaire de maîtriser

43 Whitehead, Russel, en 3 Vol., Cambridge University Press ; la bible de la logique.

44 Cf. J.Hopcroft, J.Ullman, *Formal languages and their relation to automata*, Addison-Wesley, 1969.

45 C'est ce qu'essaye de faire D.Hakehurst dans sa thèse, *Model translation : a UML-based specification technique and active implementation approach*, University of Kent.

46 Cf. *Handbook of theoritical computer science*, vol. B, MIT Press, 1990 ; entièrement consacré aux modèles formels et à la sémantique.

47 Voir par exemple : JR.Abrial, *The B-Book*, Cambridge University Press, 1996, qui contient toute une partie très intéressante consacrée aux machines abstraites.

correctement les techniques de traduction des langages et d'avoir un minimum de connaissance sur la théorie des langages.
De façon plus concrète, on peut représenter une MA par le schéma ci dessous :

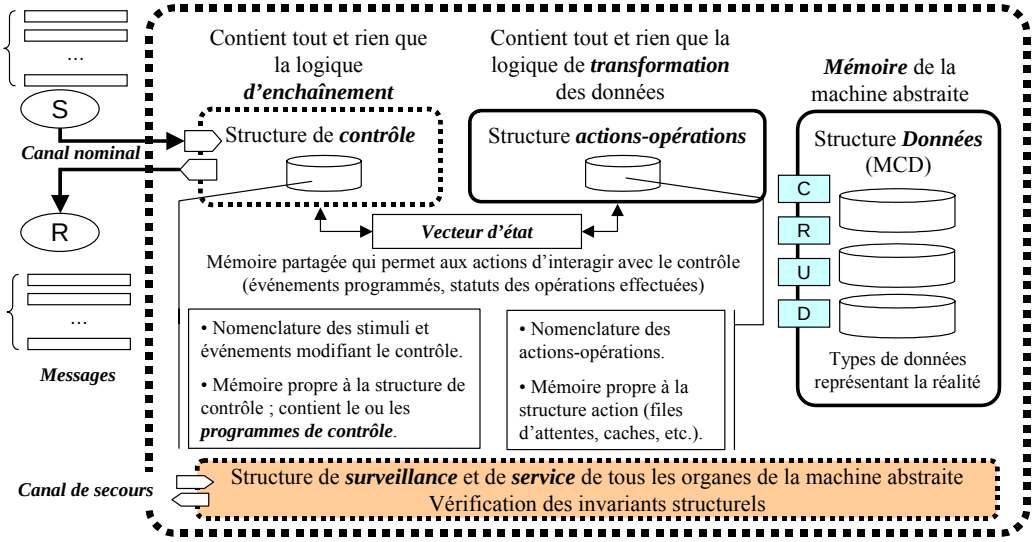


Figure 4.1 : Anatomie d'une machine abstraite – Organes fonctionnels

On retrouve les 4 organes principaux de toute machine :

a) La structure de contrôle qui contient le programme à exécuter. Notons qu'à ce stade on ne précise pas les modalités d'introduction et de représentation de ce programme.
b) La structure actions-opérations qui contient le jeu d'instructions de la machine, c'est-à-dire ce que la machine sait faire, et comment elle va le faire (caractéristiques non-fonctionnelles). Les modalités d'appels des actions-opérations définissent le langage, interne de la machine. Chacune des Actions-Opérations est un type abstrait de données (TAD).
c) La structure de données qui contient toutes les données gérées par la machine, ce qui implique à minima une nomenclature de types, un mécanisme d'adressage, une gestion de ressources.
d) La structure de surveillance et de service prend en charge tout ce qui concerne l'intégrité et la qualité du service rendu par la machine. Les invariants qui définissent l'intégrité de la machine (c'est-à-dire « sa » vérité) sont vérifiés chaque fois que cela est nécessaire.

Chacun de ces organes peut disposer de paramétrages qui leur sont propres, sous le contrôle de l'organe de surveillance.
Le schéma ci-dessous montre quelques aspects du paramétrage concernant la structure de contrôle et la partie programme.

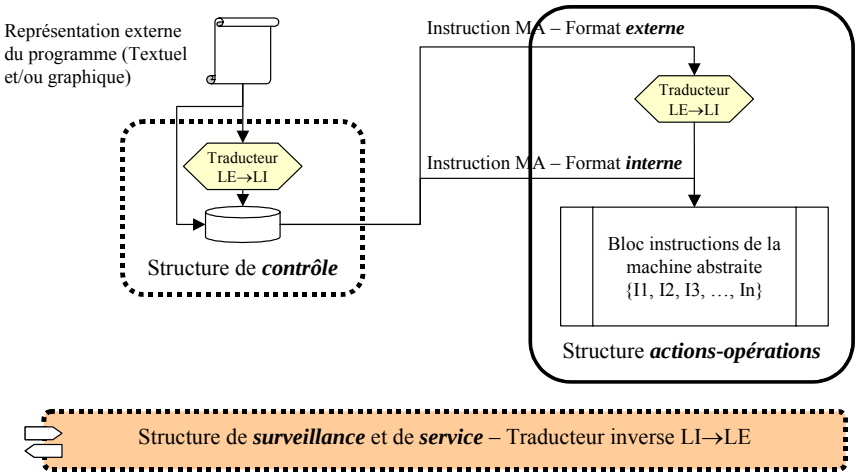


Figure 4.2 : Paramétrage de la machine abstraite.

La traduction du langage externe LE en langage interne LI peut se faire à différents niveaux, soit à la compilation, soit à l'édition de lien dynamique, soit à l'exécution au dernier moment.
C'est exactement ce qui se passe dans une machine SQL, avec le langage SQL qui peut être compilé à l'avance, ou compilé à la volée lorsque la commande SQL est construite dynamiquement. Ce mécanisme de grande puissance permet une adaptabilité bien plus fine par rapport à la réalité, mais le prix est un accroissement de complexité qu'il est indispensable de payer — tests et intégration sont plus difficiles — si ont veut garantir un niveau de qualité.

Représentations concrètes et représentations abstraites

Von Neumann, dans sa réflexion sur la structure des premiers ordinateurs, avait immédiatement compris que la manipulation de l'information selon qu'elle est opérée par un acteur humain ou par l'organe d'une machine nécessitait des modes de représentation adaptés aux uns et aux autres. C'est ce qui l'a conduit à proposer pour la machine de l'IAS à Princeton (*Electronic Computer Projet*, dont est issue l'architecture dite de Von Neumann) une représentation binaire de l'information, non pas que le code binaire ait des vertus particulières, mais simplement parce que les organes physiques (lampes triodes au départ, puis transistors) sont plus simples à réaliser. En toute théorie, on sait que si la physique nous avait permis de réaliser des composants à 3 états, cela aurait été encore plus économique (il aurait fallu une représentation en base 3), ce qui, comme on le sait, n'est pas physiquement possible avec les composants électroniques que la physique nous propose (NB : ce sera différent dans un ordinateur quantique, quand on saura les fabriquer !).
Pour ce qui concerne l'acteur humain, notre univers culturel est façonné par le texte écrit (à une dimension) et par les schémas à deux dimensions très commodes à représenter sur une feuille papier. Peut-être, dans le futur, grâce en particulier aux ordinateurs, pourrons-nous manipuler la 3D, voir la 4D pour le référentiel spatio-temporel. L'ordinateur, quant à lui, est un être mono-dimensionnel.
Les mathématiques, plus particulièrement la géométrie et la théorie des ensembles, nous enseignent qu'il est possible de faire correspondre ces diverses représentations les unes dans les autres, avec les problèmes de cardinalité et de topologie que l'on connaît. C'est la raison profonde pour laquelle il est si difficile de faire de la géométrie dans un ordinateur. Le problème n'est pas que celui de la puissance de traitement, il est encore plus celui de la représentation des êtres géométriques. L'œil humain perçoit

parfaitement les enveloppes d'une famille de droites et/ou de plan (cf. par exemple les surfaces caustiques, en optique, que chacun peut voir dans sa tasse à café), très difficilement accessibles à l'ordinateur car il faut manipuler les représentations intrinsèques des courbes et des surfaces (rayon de courbure, cercle osculateur, torsion, tenseur, etc.).
La géométrie descriptive et les perspectives nous montrent comment représenter la 3D en 2D, mais avec au passage, la perte de propriétés métriques ou la perte du parallélisme, d'ou une déformation de la réalité.
Le problème des représentations est au cœur de la technologie des compilateurs et des SGBD. On pouvait s'en douter. Dans le domaine des langages de programmation et des compilateurs, le tournant décisif a été accompli à l'occasion des travaux concernant la compilation du langage PL1 (début des années 70s). Deux notions fondamentales ont été dégagées :

a) La notion de syntaxe concrète du langage, qui est la notation familière du programmeur ; elle s'exprime en BNF, avec parfois des diagrammes de type automates (voir les premières spécifications du langage PASCAL).
b) La notion de syntaxe abstraite, qui est la notation familière de l'architecte du compilateur ; elle peut également s'exprimer en BNF (pas très intéressant), mais la représentation la plus pratique pour l'architecte est une représentation à l'aide d'arbres. Dans l'univers de la compilation, tout est arbre et forêt ; et tout est en binaire.

Ce qui est fondamental est de pouvoir passer de l'une à l'autre forme, dans les deux sens. Ce faisant, le programmeur travaille avec les modes de représentation qui lui sont le plus familier ; le compilateur déploie toute une algorithmique fondée sur les manipulations d'arbres. Chaque acteur est à son optimum.

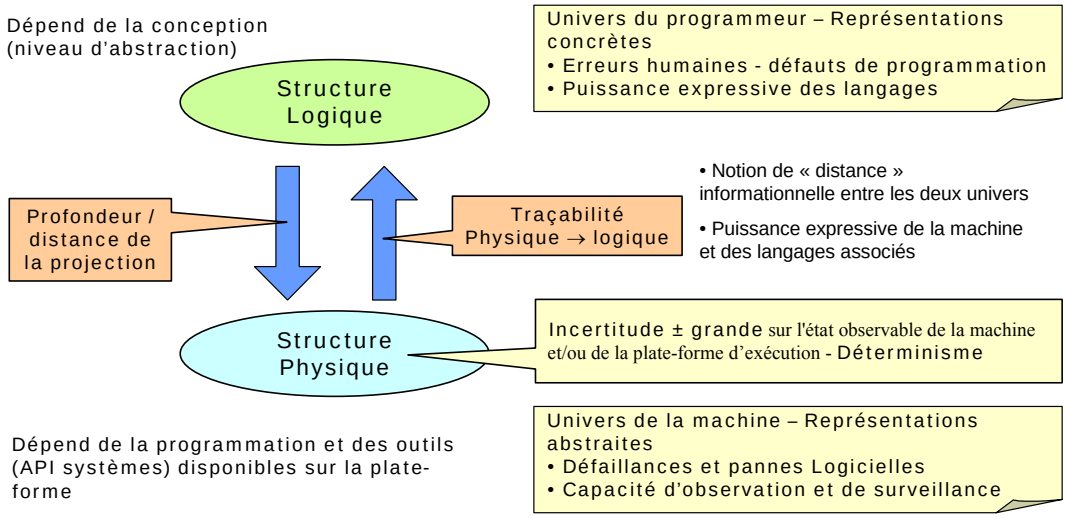


Figure 4.2a : Transformation concret - abstrait

Dans le domaine des données, les représentations ont été très fortement contraintes à l'origine, par les capacités des machines. Même aujourd'hui, avec la puissance disponible sur les UC les plus récentes, il y a toujours des limitations très importantes. Le problème du passage à l'an 2000 est encore dans toutes nos mémoires : il montre à quel point ce problème est prégnant dans toutes les manipulations d'information, même les plus simples comme les dates.
Idéalement, on pourrait ramener toutes les représentations utilisées à des chaînes de caractères ; c'est ce qu'on fait en XML. Cependant, aucun architecte d'UC sérieux

n'envisage de développer un hardware orienté chaînes de caractères ; cela a d'ailleurs été fait, et abandonné pour cause de performance rédhibitoire. Comme Von Neumann nous l'avait dit, il y a plus de 50 ans, la machine ne fonctionne bien qu'avec le langage binaire, et des unités d'alimentations (i.e. les largeurs des différents bus de la machine) qui sont des puissances de 2 : 32, 64 ou 128 bits. Vouloir utiliser des formats caractères, reviendrait à diviser la performance actuelle des UC par un facteur 100 à 1000, sans compter les problèmes de performance des entrées-sorties. Autant dire une régression inacceptable.
Si l'on veut interopérer, tant au niveau des acteurs humains, qu'au niveau des acteurs non humains, en préservant raisonnablement les capacités de traitement des plates-formes telles qu'elles sont, compte tenu des contingences du hardware, il faut traiter le problème des représentations une bonne fois par toute en distinguant soigneusement format(s) interne(s) et format(s) externe(s).
La notion la plus fondamentale est celle de système, comme il en a été question ci-dessus. Du point de vue des représentations, on peut schématiser un système comme suit :

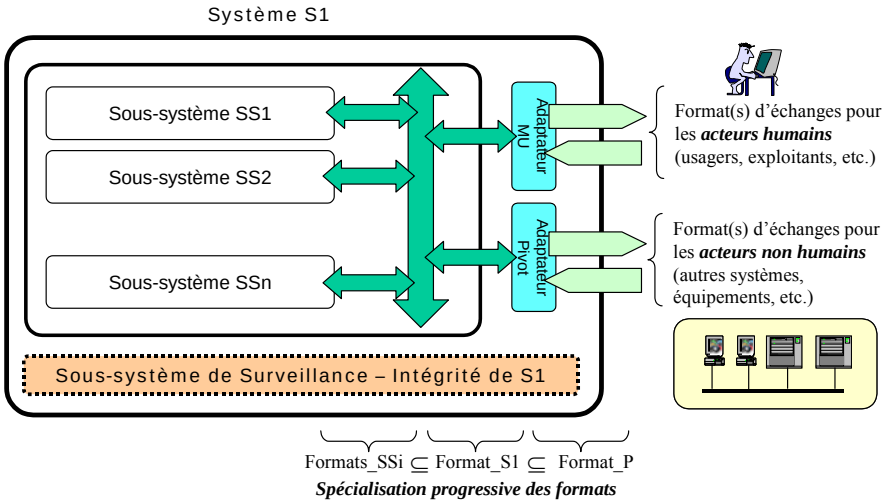


Figure 4.3 : Frontières du système – Conversion des formats d'échanges.

Dans ce schéma, la structuration en système et sous-systèmes dépend de la taille des différents organes. Pour des systèmes de grande taille[48], il peut être nécessaire de faire apparaître les 3 niveaux d'adaptation : Fédération – Système – Sous-systèmes, et ceci pour des raisons de structuration des projets correspondants. Les trois sphères de contrôles ainsi identifiée définissent des frontières qui doivent être associées à des critères d'appartenance de ce qui est DANS et HORS de la sphère de contrôle de l'entité système considérée.
Ceci montre que dans la problématique des étapes de conversions et d'adaptations, l'ingénierie du système prend sa part. Ce qui est nécessaire et suffisant est de se mettre d'accord sur la sémantique. Les représentations sont purement conjoncturelles, puisque la sémantique commune permet la traduction des formats les uns dans les autres. Le

[48] Se reporter aux normes de l'ingénierie système, comme la IEEE 1220 ; cf. JP.Meinadier, *Intégration et ingénierie système*, Hermès.

reste est affaire de capacité et de maturité des équipes, mais requiert, a minima, un niveau de maturité 3-4 du CMM/CMMI[49].
Notons que le langage graphique utilisé par les acteurs humains pour interagir avec les systèmes est partie prenante de l'interopérabilité, via le modèle utilisateur (MU) qui lui est associé ; nous ne faisons que mentionner ce point qui est un problème en soi[50].
On peut édicter la règle d'ingénierie suivante :

Règle :
Pour toute structure intervenant dans un échange, il est fondamental de distinguer les modes de représentations selon le point de vue des acteurs concernés, et de spécifier les traducteurs qui permettent de passer d'une forme à l'autre, dans les deux sens.

La traduction réversible des types de données

Dans tout système d'information, et ce dès l'origine des premières applications de gestion, le problème des conversions de types a été l'une des préoccupations centrales des concepteurs de langages informatiques. Par exemple, dans la fonction édition de résultat (cf. la notion de `REPORT WRITER` en COBOL, et les instructions `FORMAT` en FORTRAN), il faut convertir des types internes de la programmation (le modèle logique-physique dans notre terminologie) dans des types compréhensibles par un œil humain (le modèle de données utilisateurs MDU, qui est un modèle conceptuel) associés à des règles de présentation (couleurs, structuration de l'écran ou du rapport d'édition, etc.).
Le premier langage dans lequel la problématique a été posée dans toute sa généralité, hors des cas particuliers de l'édition, a été le langage PL1 ; et ce fut un échec retentissant. Comme très souvent dans l'histoire des sciences et des techniques on apprend plus de l'analyse de ses échecs, que par toute autre méthode ; mais cela demande courage et lucidité, vertus de plus en plus rares.
Le langage PL1 autorisait des constructions d'expressions dont les variables pouvaient avoir des types différents. A charge du compilateur d'effectuer les bonnes conversions au moyen d'une matrice de conversions, analogue aux matrices N2 utilisées en ingénierie système. C'était très compliqué à implémenter, avec des règles parfois complètement arbitraires : une belle prouesse technique dont la caractéristique U de FURPSE n'avait pas du tout été analysée.
A l'usage, et ce très rapidement, plusieurs phénomènes émergents, non prévus par les concepteurs du langage, se sont manifestés :
a) Dans la plupart des cas une conversion implicite de types est, en fait, le résultat d'une erreur de programmation ; en conséquence, la conversion va masquer le défaut, ce qui est l'inverse de ce qu'il faudrait faire. Ce que les concepteurs de langage avaient mal apprécié, c'est que derrière l'entité informatique manipulée, il y a une « grandeur » au sens sémantique du terme. La chaîne de bits manipulée par le compilateur a un sens pour l'usager, mais ce sens est inaccessible au compilateur. Conclusion, il vaut mieux rendre les conversions explicites.
Pour bien comprendre la nuance, rappelons nous ce qu'est un type en physique, c'est-à-dire l'équation de dimension de la grandeur. La notion de type, au sens informatique du terme, a une parenté profonde avec celle d'équation de dimension associée aux

[49] Capability Maturity Model, inventé par W.Humphrey au SEI, et son extension à l'ingénierie système CMMI.
[50] Cf. le Mil-std-2525B, déjà cité.

grandeurs physiques. Une équation de dimension permet de faire des contrôles sémantiques par rapport aux grandeurs physiques fondamentales qui constituent le système d'unités (MKS, CGS, etc.) qui sert à représenter numériquement ces grandeurs. Par exemple, un travail (i.e. en Joule ou KWH) s'exprime par le produit d'une force (i.e. $F=M\times\gamma$) et d'une distance L sur laquelle la force a été appliquée, soit :

$$W = F \times L = M \times \frac{L}{T^2} \times L = M \times \left(\frac{L}{T}\right)^2 = M \times L^2 \times T^{-2}$$

NB : on reconnaît la forme de l'énergie cinétique $E = \frac{1}{2}mv^2$.

Toute manipulation mathématique d'une grandeur qui représente un travail ou une énergie doit être conforme à ce schéma logique, faute de quoi le calcul n'a pas de sens.
On ne peut additionner ou soustraire que des grandeurs de même dimension, faute de quoi l'opération est incohérente. Dans une formule de type A=B (assignation, ou comparaison, ou équivalence), les dimensions de A et de B doivent être identiques. Le type est à la fois un nom qui véhicule de la sémantique et une règle d'interprétation de la mémoire qui sert à stocker l'entité informatique.
Mélanger dans une même expression, des entités de types différents provient la plupart du temps d'erreurs de manipulation concernant la sémantique des entités manipulées ; pour parler vulgairement on a additionné des choux et des poulets. Plutôt que de convertir, il faudrait éditer un message d'erreur, ou solliciter un opérateur humain, ou.. etc.

b) Pour convertir correctement les grandeurs, il faut connaître l'usage précis de la donnée, c'est à dire son type abstrait complet. Par exemple, en comptabilité, l'usage est de faire des arrondis par défaut et/ou par excès selon les contextes ; en fiscalité, on fait toujours des arrondis par défaut, à l'unité inférieure.
Pour les nombres flottants, la sémantique est en apparence celle des réels, mais ceci est faux comme chacun sait car la « précision » du nombre flottant se consomme comme une ressource au fur et à mesure de l'avancement des calculs, d'où des défaillances de code extrêmement perverses que les concepteurs de systèmes spatiaux comme le GPS connaissent très bien.
Pour les entiers, c'est encore différent. Des opérations comme $a = b + c$, ou $a = b \times c$ peuvent déclencher des dépassements de capacités : que faire dans le programme ? Dans la généralité (par exemple, du point de vue du compilateur, qui est nécessairement « context-free »), on ne peut rien faire de vraiment sensé : tronquer le résultat, mais quels bits faire disparaître ? émettre un warning, mais vers quels acteurs destinataires ? émettre une exception, mais vers quel pilote (« driver ») d'exceptions ? de plus, cela peut dépendre de l'ordre des opérations ; par exemple : $a = b \times c - d$ peut ou non lever une anomalie à l'exécution, si l'on comprend l'expression comme une suite : $a = b \times c$; $a = a - c$; dans ce cas, cela veut dire que parfois, l'opération n'est plus commutative !
Pour respecter cette règle apprise dès le plus jeune age, il faudrait effectuer les calculs intermédiaires avec des variables de précision telle qu'elles ne peuvent pas générer l' « overflow ».
En conclusion, on voit qu'il est parfaitement illusoire de vouloir convertir en toute généralité, en préservant un invariant sémantique qui dépend du contexte d'emploi, donc non « *context-free* » par définition.

c) L'erreur la plus subtile est cependant une erreur logique. Dans les premiers langages de programmation COBOL, FORTRAN, PL1, C (et quelques autres), un type de donnée

est en fait une description de la mémoire qui sera allouée à la donnée. Pour la machine (et pour le compilateur) le sens de la donnée est sa représentation mémoire et les opérations autorisées sur ce type particulier ; d'où la question qui tue : est-ce que ce sens est celui qui est dans la tête de l'utilisateur du programme ? Et la réponse : pas toujours !

Pour parler comme Frege, on a confondu le sens et la dénotation, et pour parler comme les linguistes (cf. Saussure) on a confondu le contenant et le contenu.

Calculer sur l'un, n'est pas toujours équivalent à calculer sur l'autre. Car en assimilant l'un à l'autre, on incorpore subtilement la machine sous-jacente dans le modèle conceptuel ! C'est la raison pour laquelle les architectes du système MULTICS[51] avaient séparé les calculs d'adresses (avec des registres de base, permettant entre autre d'implémenter le modèle de sécurité des accès et la protection par « ring ») et les calculs arithmétiques (avec les registres généraux). La bonne réponse à ce dernier problème a été l'invention du concept de type abstrait de données et des langages dits « fortement typés » comme PASCAL , et surtout Ada.

Dans ces langages (et ceux qui ont suivi, comme les langages objets) on distingue soigneusement, au moins au niveau de la syntaxe, le type d'une donnée et l'instance de la donnée, ce qui permet de déclarer séparément les types et les instances (i.e. les variables), alors que dans la génération de langages précédente, tout était confondu, y compris les règles d'allocation et de portée des variables (i.e. ce qui est privé, publique, avec toutes les variantes possibles, etc.).

Par ailleurs, en théorie du moins, le sens complet est définie par la nature les opérations effectuées sur la donnée, ce qui, dans la masse des instructions programmées, exige de répertorier toutes celles qui directement, ou indirectement (comme les exceptions levées), ont affaire à ce type particulier.

Le problème de la levée des exceptions est redoutablement complexe ; il convient de l'aborder avec lucidité et modestie car les meilleurs architectes s'y sont cassés les dents. Le traitement de l'exception dépend généralement du contexte dans lequel l'exception a été levée, ce qui veut dire qu'il faut également définir dans le langage des règles de portée des exceptions, statiquement et dynamiquement, pour spécialiser correctement les traitements correspondants. C'est ce qu'avait essayé de faire les architectes du langage PL1 (instruction `on condition` …), avec là encore l'échec comme résultat.

Actuellement, les meilleurs mécanismes disponibles dans les langages, sont ceux de Ada 95, mais ils sont encore très loin de refléter la variété des situations sémantiques, ce qui veut dire que le travail sémantique à la charge du programmeur reste irremplaçable.

La notion de gravité associée au traitement correspondant de l'exception est purement sémantique, il ne faut donc pas s'étonner d'avoir quelques difficultés à la faire rentrer dans le moule général d'une syntaxe.

En raisonnant objet, cela reviendrait à pouvoir associer au contexte d'emploi de l'objet, un jeu de méthodes particulières dépendantes de l'état courant du traitement. En environnement distribué, avec les appels de services distants, on mesure la difficulté à spécifier correctement un tel mécanisme, et à garantir sa sémantique opérationnelle dans toutes les situations possibles. C'est un domaine où les middleware d'intégration[52]

---

[51] Cf. Organick, *The multics system*, MIT Press. Développé avec l'aide du MIT, le système a été commercialisé par Bull vers la fin des années 70s. Cf. également une note interne Bull de W..Kohlbrenner, *Overview of NPL synchronization hardware*, disponible aux archives Bull.

[52] Rappelons, pour mémoire, que la seule société française à offrir un tel progiciel est la société SOPRA, via sa filiale AXWAY.

généralistes (les EAI) touchent leurs limites, car par définition l'ensemble des contextes est un ensemble ouvert[53].

L'introduction des types abstraits tels que nous les connaissons aujourd'hui est probablement l'un des plus grand progrès réalisés en matière de langages de programmation, après l'invention des langages de 3ème génération. A t-on pour autant fait disparaître le problème ? Malheureusement non !

Mais on a clairement distinguer les rôles :

- C'est au programmeur de typer correctement les entités du réel, qu'il souhaite manipuler ou celles qui en sont dérivées, avec des opérations (en objet, des méthodes) dont il a la responsabilité entière.
- C'est au compilateur d'implémenter correctement la sémantique associée aux types et aux opérations, ce qui va généralement impliquer des vérifications dynamiques.

Par exemple, pour une expression comme $a = b \times c$ avec $\{a, b, c\}$ ayant des plages de valeurs comprises entre $[-20 ; +50]$, dont le compilateur a décidé d'implémenter les variables correspondantes avec des entiers signés courts (15 bits, et un bit de signe), on peut générer une vérification avant d'assigner le résultat du calcul $b \times c \xrightarrow{dans} a$.

Mais si le programmeur, pour ne pas être embêté par les messages d'erreurs, et pire, pour faire croire à ses collègues qu'il sait écrire du code « performant », il a tout typé en entier court, on est exactement ramené à la situation précédente.

Notons que si le programmeur a cru bon d'écrire, comme précédemment, $a = b \times c$; $a = a - c$; la situation est encore plus confuse, car cela peut dépendre des algorithmes d'optimisation mis en œuvre par le compilateur. Un compilateur moyennement intelligent s'apercevra que l'écriture précédente est équivalente (du point de vue des types, c'est parfaitement exact) à : $a = b \times c - d$ qui ne doit pas lever l'exception « overflow » !

C'est un cas très simple, où l'on voit que la sémantique opérationnelle dépend des options de compilation. D'où une règle d'ingénierie du typage et de l'objet.

Règle du typage des entités

Les langages fortement typés (Ada 95, JAVA pour n'en citer que deux) donnent la possibilité de programmer proprement, et de ce fait sont préférables à tout autre langage, mais ne garantissent en aucune façon que la sémantique comportementale de l'objet correspondant est exactement celle de l'entité su monde réel dénoté par cet objet. Les choix de modélisation ne concernent que le concepteur, personne d'autre, et surtout pas un quelconque générateur de code qui ne fait que traduire fidèlement ce que le programmeur a écrit. Le code généré doit être immuable (c'est une boite noire), et ne jamais être retouché par un programmeur.

Si cette règle n'est pas respectée, le modèle qu'est le code source n'est pas conforme à la réalité à modéliser ; au sens strict, ce n'est donc pas un bon modèle, et il sera rapidement obsolète, conformément à la règle d'obsolescence vue ci-dessus.

Un autre problème lié au typage, concernant la représentation des ensembles, est intéressant à analyser dans l'un et l'autre des systèmes de représentation.

Dans un langage comme PL1, on peut représenter un ensemble de diverses façons : par la valeur d'une variable numérique, par une chaîne de bits, par une chaîne de caractères. La représentation par chaîne de bits a eu beaucoup de succès auprès des programmeurs

[53] Cf. J.Printz, *Programmer en univers incertain*, déjà cité.

PL1 car cela permettait de manipuler le rang de un ou plusieurs éléments de l'ensemble, tout en gardant la possibilité de faire des opérations booléennes, réservées aux seules chaînes de bits (le programmeur dispose, en une seule représentation, de l'extension de l'ensemble, d'un ordre, et d'opérations de manipulation de l'ensemble via les opérateurs booléens de la logique classique.
NB : Dans les machines, la chaîne de bit est le format de donnée le plus neutre qui soit, celui à partir duquel on peut dériver tous les autres.
Avec les langages fortement typés, ce genre de « cuisine » est assez difficile à faire, pour ne pas dire impossible. Si l'on prend le cas d'un type énuméré, on peut le représenter soit par la valeur d'une variable, soit par une position dans une chaîne de bit : c'est un choix de compilation ; on peut restreindre les opérations possibles comme le fait Ada ou JAVA, c'est-à-dire le minimum, en faisant en plus l'hypothèse que le type est immuable. La question qui fâche est : est-ce que cette sémantique est celle correspondant à des situations du monde réel ? Prenons deux cas, au cœur des problèmes rencontrés quand on veut faire interopérer des systèmes.

Cas N° 1 : les codes d'affiliation
Dans le Mil-std 2525B déjà mentionné, on caractérise certaines entités de la réalité par un code appelé « code d'affiliation », qui peut prendre 10 valeurs, soit {P - Pending, U - Unknown, A - Assumed friend, F - Friend, N - Neutral, S - Suspect, H - Hostile, J - Joker, K - Faker, O - None specified}. Cette affiliation est le résultat d'une certaine analyse/synthèse faite par un ou plusieurs acteurs du système, mais il est évident que dans certaines situations on aimerait graduer le concept H-Hostile, selon que l'entité hostile dispose d'un couteau, d'une kalachnikov, ou d'une bonbonne de gaz sarin, ou … (c'est une liste ouverte !). La rigidité du typage énuméré va donc créer une vraie difficulté de représentation, et en fait interdire l'usage de ce typage.
NB : Notons que l'on ne peut pas s'en tirer simplement en introduisant un des types complémentaires, car on introduit alors des dépendances fonctionnelles entre les types qu'il faudra gérer (en fait, c'est la notion de transaction qui s'introduit de façon subreptice).
On va donc passer d'une cuisine, certes pas très propre, mais tout de même mangeable, à quelque chose de peu ragoûtant, et on peut faire confiance aux programmeurs pour « inventer » des astuces parfaitement indigestes qui violeront toutes les règles de bonne programmation ; d'ingénieur, on est passé au stade du bricoleur.

Cas 2 : les codes de retour (exceptions et/ou événements) générés par les appels de service
Dans une programmation bien faite, l'appel d'un service se traduit, lors du retour à l'appelant, par le positionnement d'un code de statut qui définit l'état du résultat obtenu : {FAIT, NON FAIT, …} avec en général tout un ensemble de modalités que peut concerner le FAIT comme le NON FAIT; par exemple l'appel au service d'ouverture d'un fichier peut se traduire par la modalité FAIT-DEJA-OUVERT, ce qui est une information généralement intéressante pour l'appelant car cela veut dire qu'un autre utilisateur travaille avec ce fichier. Le problème des codes retour est qu'il est impossible de les considérer comme des ensembles immuables, car d'une version à l'autre, d'une machine à l'autre, les modalités peuvent être plus ou moins riches, et de toute façon différente car standardiser à ce niveau revient à choisir une machine.
Cet exemple n'est d'ailleurs qu'une nouvelle illustration, sur un cas particulier, de la difficulté, d'un point de vue sémantique, du traitement des exceptions. Il y a des exceptions qui vont correspondre à des évènements liées au métier, d'autres qui seront

liées à des représentations et d'autres encore qui seront spécifiques aux plates-formes en sachant que les frontières entre ces différents univers, ne sont jamais parfaitement nettes.

Dans ce cas, il est quasiment impossible d'utiliser le type énuméré pour représenter cette sémantique, alors que cela paraissait naturel en première analyse. Ce qui crée la difficulté est que le codage effectué via le type énuméré est plus qu'un simple codage ; ce codage est porteur de sémantique, et l'opération, à première vue anodine :

V1: Type_A = V2 : Type_B

est en faite une véritable transduction[54], qui devra être détaillée au cas par cas, selon la matrice des cas possibles correspondant au produit cartésien $\{a_1, a_2, a_3, \cdots, a_m\} \times \{b_1, b_2, b_3, \cdots, b_n\}$ avec peut-être un appel vers l'utilisateur pour certaines combinaisons contextuelles.

N.B. une analyse plus fine, nous amènerais à évoquer la loi de la variété indispensable, de R. Ashby[55], particulièrement appropriée ici puisqu'il s'agit d'information (à tous les sens du terme !) ; nous n'en dirons pas plus dans cette monographie.

Pour bien montrer la différence avec un code véritable, on peut examiner les différentes façons de coder des arborescences, largement utilisées dans les compilateurs. Par exemple : l'expression $a = b + c \times d$ (notation infixe habituelle) est strictement équivalente à $abcd \times + =$, notation post-fixe, inventée par le logicien polonais Lukasiewicz pour éliminer les () des expressions logiques, ou à $= a + b \times cd$ (notation pré-fixe, du même inventeur), quelle que soit la complexité de l'expression. On peut passer de l'une à l'autre représentation dans tous les cas de figure. Dans ce cas la transduction dégénère en simple traduction, et il n'y a jamais besoin de faire appel à l'usager pour savoir comment traduire et lever les difficulté sémantique. La syntaxe proposée coïncide exactement avec la sémantique.

Avec les bases de données, et en particulier dans la foulée des travaux ayant aboutis au rapport ANSI-SPARC, (élaboré entre 71 et 75), une étape supplémentaire à été franchi dans l'ingénierie des conversions et changements de formats. C'est dans ce rapport que les notions de schémas et de sous-schémas ont pris toute leur importance, ainsi que celles de méta-data (on disait, alors, dictionnaire de données, car on était sobre dans la terminologie).

La notion de schéma permet de définir de façon rigoureuse des mappings entre les enregistrements (entre les `record` de COBOL et du modèle CODASYL, ou entre les `t-uple` du modèle relationnel) d'une base données, car par définition il n'y a jamais de recouvrement entre deux enregistrements qui sont nécessairement disjoints.

Là encore, PL1 avait essayé d'ouvrir la voie aux opération sur les agrégats de données, avec en plus des notions d'agrégats de taille variable (XML avant la lettre !) mais en n'attachant pas assez d'importance aux manipulations sur la mémoire persistante (i.e. les disques) et à celles en mémoire volatile (i.e. la mémoire centrale). La même opération en mémoire centrale n'a généralement pas de sens (cf. le problème de l'assignation multiple) avec des opérations ensemblistes du type :

agrégat_1 = agrégat_2

(cela revient à programmer un filtre) dont le sens approximatif était une suite d'assignation sur les attributs élémentaires des agrégats. Outre les problèmes de conversions déjà signalés, il se rajoute celui des recouvrements possibles en mémoire, ce qui est générateur d'effets de bord parfaitement incontrôlables, surtout avec des

---

[54] Voir annexe 2 ; la transduction est une notion de la théorie des automates.

[55] Cf. *An introduction to cybernetics*, Methuen & Co.

langages ayant une notion de pointeur non typé comme PL1 ou C. C'est une autre raison du rejet progressif de PL1 qui autorisait ce genre de manipulation contraire à toutes les bonnes règles d'ingénierie de la programmation. Un programmeur « moyen » était sûr de faire de grosses bêtises.
Une assignation multiple (manipulation d'agrégats) n'a de sens que si on peut l'assimiler à une transaction, ce qui permet effectivement de la décomposer en une succession d'opérations élémentaires
$\{a1 = b1\,;\, a2 = b2\,;\, a3 = b5\,;\, a4 = b3;\, etc.\}$
considérée comme un bloc respectant ACID. Mais pour cela, il faut disposer du concept de transaction intégré au langage (c'est une sémantique opérationnelle), ce qui n'est le cas d'aucun langage actuel.
NB : Ada 95 dispose d'un certain nombre de mécanismes tout à fait intéressant, mais il est douteux (pour ne pas dire certain) que le langage revienne un jour dans le jeu industriel des systèmes d'information.

Les langages qui ont le mieux (ou le moins mal !) exploités ces possibilités ont été les L4G des années 80s mais aujourd'hui il n'est reste presque rien. De plus, les architectures distribuées compliquent singulièrement un problème qui n'était déjà pas simple dans les architectures centralisées.
Sur ces sujets, voir le livre très intéressant de B. Walraet, *Programming, the impossible challenge*, North-Holland. B. Walraet fut un des collaborateurs de la société Cullinet Software, (NB : l'Oracle des années 70-80 !) aujourd'hui disparue dont le L4G ADS-On-Line (ADSO) fut probablement le meilleur L4G jamais conçu.

Règle d'ingénierie de l'architecture pilotée par les modèles
Pour aborder sereinement la problématique MDA, et éviter de réinventer la roue, il est indispensable de se replonger dans toutes ces études qui ont eu un impact industriel de première grandeur avec le modèle relationnel, et tout ce qui en est dérivé[56].

Implémenter l'interopérabilité.

La faisabilité de l'implémentation de l'interopérabilité revient à répondre à la double question : Peut-on définir une machine à inter-opérer ? Peut-on garantir la qualité d'une telle machine ? Peut-on gérer les évolutions de la machine dans le temps ?

La machine à interopérer

Pour que cette machine soit complètement définie, il faut spécifier les quatre organes de la machine abstraite définie précédemment :

1. La structure de contrôle
2. La structure actions-opérations (i.e. les services rendus par la machine)
3. La nomenclature des types reconnus par la machine
4. La structure de surveillance (invariants, règles d'intégrité, etc.)

Commençons par la structure actions opérations.

Structure Actions-Opérations

La structure Actions-Opérations est un type abstrait de données (TAD, ou ADT en anglais) au sens strict du terme. C'est un ensemble de services basiques (i.e. les « instructions » de la machine) complètement défini : tous les services sont identifiés,

[56] Voir en particulier les recensions de M.Stonebraker, *Readings in database systems* et de S.Zdonik, D.Maier, *Readings in object-oriented database systems*, tous deux chez Morgan Kaufmann.

ainsi que leurs conditions d'appels (cf. la notion de contrat) ; la nature des types manipulés est connue, ainsi que l'état de l'exécution du service (l'aspect « PERFORM »), i.e. FAIT, NON-FAIT, et toutes les modalités intermédiaires : c'est le statut de l'opération (i.e. le vecteur d'états). Cette structure est le « jeu d'instructions » accessibles à chacun des systèmes constituant la fédération. Le jeux d'instructions définit ce que la machine sait faire (aspect fonctionnel) et comment elle va le faire (aspects non-fonctionnels).
La structure externe fonctionnelle de chacun de ces services peut être schématisée comme suit :

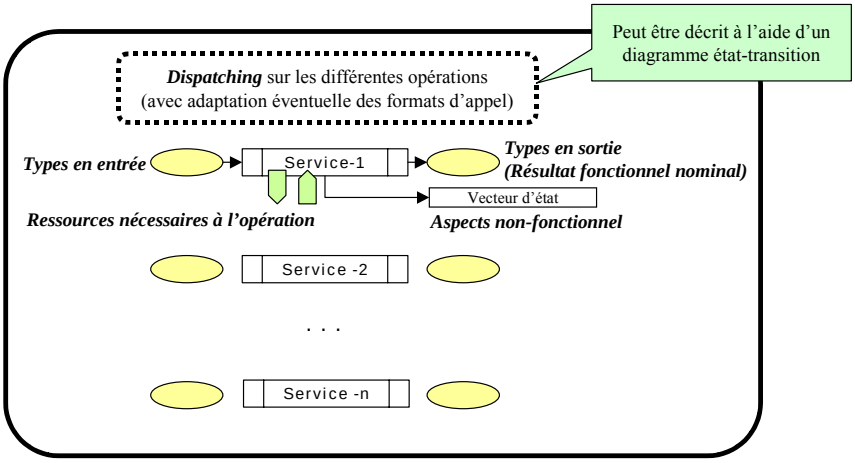


Figure 4.4 : Structure fonctionnelle des actions-opérations.

A ce niveau, on ne fait aucune hypothèse sur le déploiement des services ; ils peuvent être locaux ou distants selon les contraintes spécifiques à chacun des systèmes. Les ressources nécessaires peuvent être gérées de façon autonome par le service lui-même, ou fournies par l'appelant du service.

La nomenclature des types.

La nomenclature répertorie tous les types de données qui peuvent être utilisés par la structure actions-opérations. Pour des raisons d'extensibilité de la structure action-opérations, il faut se donner la possibilité de travailler sur deux niveaux de typage :

a) Le niveau N0 qui constitue les types primitifs qui correspondent aux opérations atomiques de la machine abstraite correspondant à la granularité la plus fine de la sémantique qu'il faut modéliser. Sur ce niveau, la machine doit être déterministe.
b) Le niveau N1 qui est un niveau construit sur le niveau N0, avec par exemple comme objectif de définir la sémantique orientée métier que l'on souhaite définir en tant que service pour l'ensemble de la fédération.

D'où une construction à deux niveaux de la structure actions-opérations, et la mise en évidence d'un niveau de paramétrage de première importance. On a des structures de ce type dans toutes les UC avec la distinction machine / micro-machine, ou les convertisseurs d'adresses dans les caches[57].
D'un point de vue sémantique, le niveau N0 sera spécialisé sur la plate-forme sous-jacente. Ce niveau N0 pourrait implémenter la correspondance PIM vers PSM, d'où la nécessité absolue du déterminisme du niveau N0, sans lequel il n'y a pas de traduction automatique possible. Le niveau N1 implémente les objets métiers. C'est ce niveau qui

[57] Cf. J.Hennesy, D.Patterson, *Computer architecture, a quantitative approach*, Morgan Kaufman.

doit être doté des facilités de paramétrage les plus puissantes (c'est à dire de programmation), car c'est lui qui permet l'adaptation aux fluctuations de la réalité.

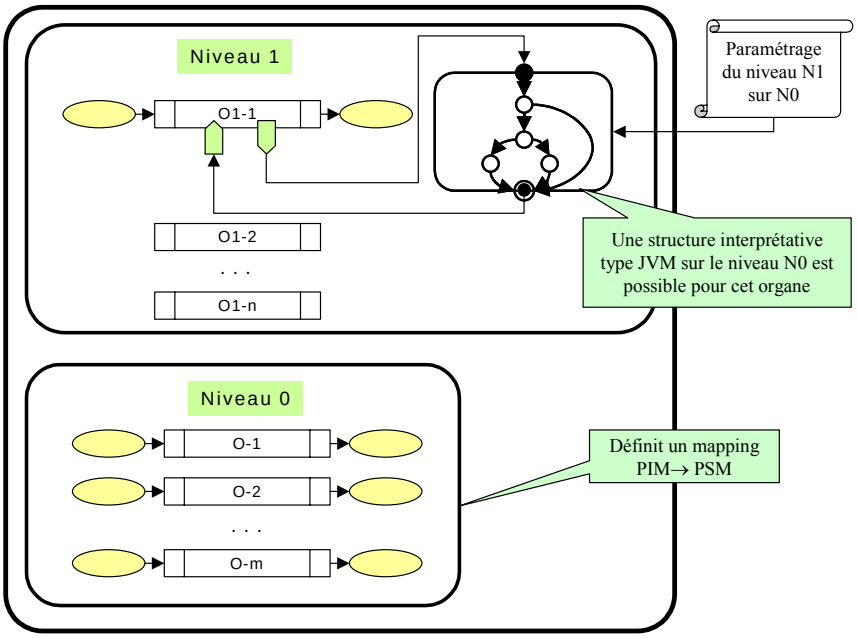


Figure 4.5 : Organisation de la structure actions-opérations.

Notons que la structure interprétative du niveau N0 sur N1 peut être décrite de façon concrète sous une forme déclarative (et non impérative) comme cela a été fait dans les SGBD entre les niveaux conceptuels et logiques dans les années 70-80s, puis dans les langages de 4ème génération au milieu des années 80s.
A la fin d'une opération, qu'elle soit de niveau N0, ou de niveau N1, l'opération restitue un résultat et un vecteur d'état précisant les modalités d'obtention de ce résultat.
La standardisation du vecteur d'état est une nécessité absolue si l'on veut spécifier rigoureusement la sémantique comportementale des opérations effectuées car elle va déclencher l'appel de telle ou telle fonction de surveillance et/ou d'intégrité.

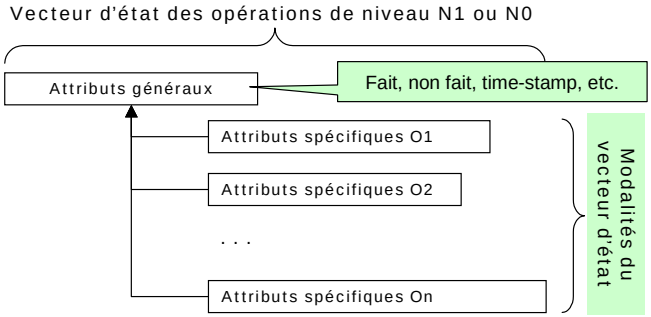


Figure 4.6 : Standardisation du vecteur d'état.

Parmi les propriétés générales véhiculées par le vecteur d'état on aura : l'état {FAIT, NON FAIT}, la date universelle (TIME-STAMP, ou estampille temporelle) et la durée de l'opération, les performances, les anomalies rencontrées (conflits d'accès, etc.), le nombre d'opérations intermédiaires faites au niveau N0, etc. Parmi les attributs spécifiques, on aura : la liste détaillée de toutes les opérations intermédiaires (identification dans l'historique de l'exécution du service), le détail des modalités de

l'échec si on est dans l'état NON-FAIT, les ressources inaccessibles, opération interrompue et/ou suspendue, etc.
Tout ou partie du vecteur d'état est transmis à la structure de contrôle pour exploitation.

La structure de contrôle.

La structure de contrôle de la machine abstraite d'interopérabilité a un triple rôle :

a) Lancer l'exécution requise par le programme de contrôle qui a été chargé dans la mémoire de la structure de contrôle.
b) Recueillir les résultats des opérations effectuées et transmettre ces résultats à l'environnement appelant.
c) Analyser le vecteur d'état et les messages de contrôle ayant pu être transmis à la machine pendant la durée de l'opération de façon à sélectionner la prochaine opération à effectuer. A tout moment, la structure de contrôle doit pouvoir mettre fin aux opérations en cours, si des évènements prioritaires qui le nécessitent sont survenus. Dans ce cas la machine doit être arrêtée dans un état cohérent qui ne met pas en cause son intégrité

La structure simplifiée d'un cycle de contrôle peut être schématisée comme suit :

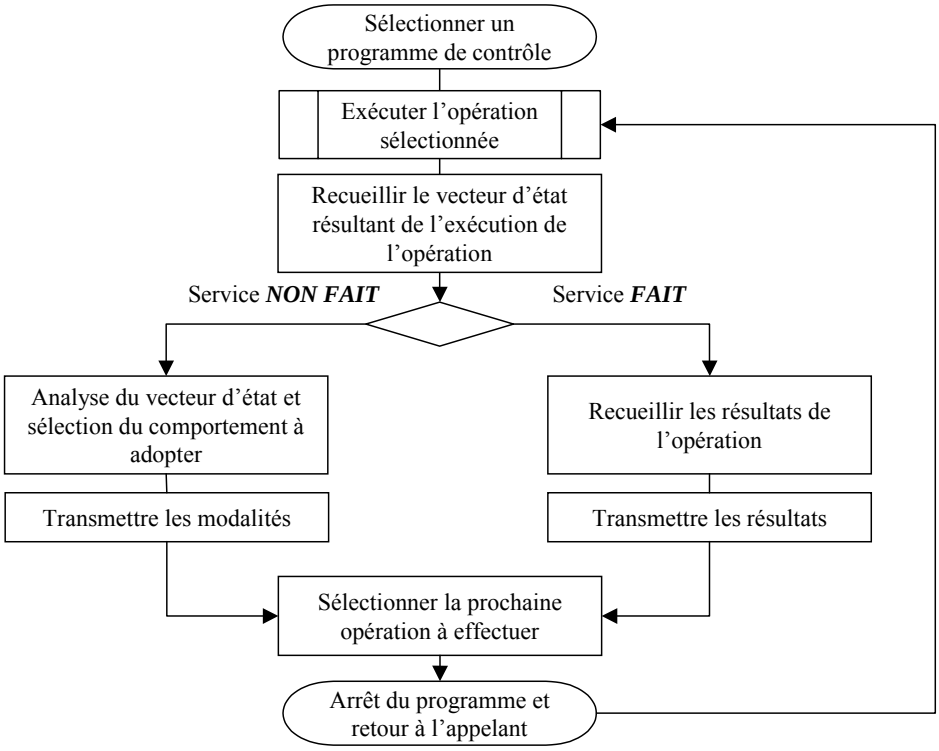


Figure 4.7 : Structure d'un cycle de contrôle.

Il faut prendre en compte le fait que le service effectué est soit local, soit distant car la sémantique du contrôle sera différente dans chaque cas.

La structure de surveillance.

Cette structure gère tout ce qui est susceptible d'altérer le bon déroulement des opérations. Elle construit le vecteur d'état. En fonction des observations effectuées sur les différents organes de la machine, y compris sur elle même, elle peut notifier à la structure de contrôle d'interrompre ou suspendre les opérations en cours.
En cas de panne générale de la machine abstraite, elle doit disposer d'un canal de secours permettant de notifier à l'environnement que la machine n'est plus en état de

rendre les services attendus ; ce canal doit être distinct du canal nominal pour des raisons de sûreté.

La structure de surveillance gère l'ensemble des tests permettant de s'assurer que toutes les opérations sont vérifiées et validées, ainsi que toutes les sondes nécessaires à l'observation des organes. Le dispositif d'observation est lui-même paramétrable en fonction du niveau de maturité de la machine.

Gérer les évolutions fonctionnelles de la machine abstraite

Telle que spécifiée, la machine abstraite d'interopérabilité peut évoluer sur différents axes.

Créer de nouvelles opérations sur les types existants

La structuration en deux niveaux facilite ce type d'extension. En particulier on peut envisager d'enrichir, statiquement ou dynamiquement, la bibliothèque des opérations possibles, telle que décrite dans la Figure 4.5.

Créer de nouveaux types de données et les opérations correspondantes

Si les types de niveau N0 sont invariants, l'extension peut être réalisée en utilisant le mécanisme de paramétrage spécifié pour le niveau N1.

Enrichir les types de données existants et enrichir les opérations existantes

C'est l'opération d'extension la plus délicate à réaliser car les projets gérant les systèmes utilisant les services de la machine abstraite peuvent ne pas être à même d'effectuer les extensions requises par certains d'entre eux.

La structure actions-opérations doit être organisée pour pouvoir gérer les différents niveaux de norme requis pour la mise en œuvre de la fédération.

La structure fonctionnelle générique d'une opération susceptible d'évoluer doit être revue comme suit :

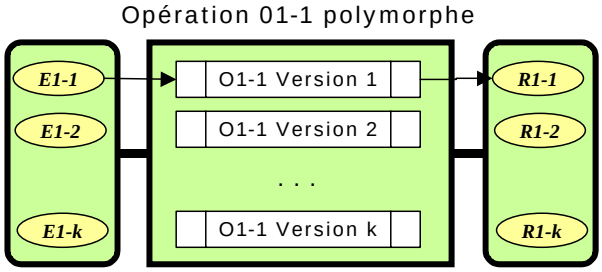


Figure 4.8 : Extensibilité des opérations.

C'est une opération polymorphe, mais avec une sémantique de polymorphisme très précise qui est celle de l'évolution des normes vue au chapitre 3 ci-dessus.

Esquisse d'un langage pivot complet

Dans les chapitres précédents nous avons vu un certain nombre d'éléments constitutifs du langage et de l'architecture pivot, en particulier à travers les figures 2.10, 2.13, 3.5, 3.6, 4.3, 4.4, 4.5. Nous allons maintenant ébaucher les grandes lignes d'un langage pivot permettant de programmer complètement les machines abstraites constitutives de l'architecture de l'interopérabilité.

Vue d'un système particulier, le potentiel d'interopérabilité est décrit conformément au schéma suivant :

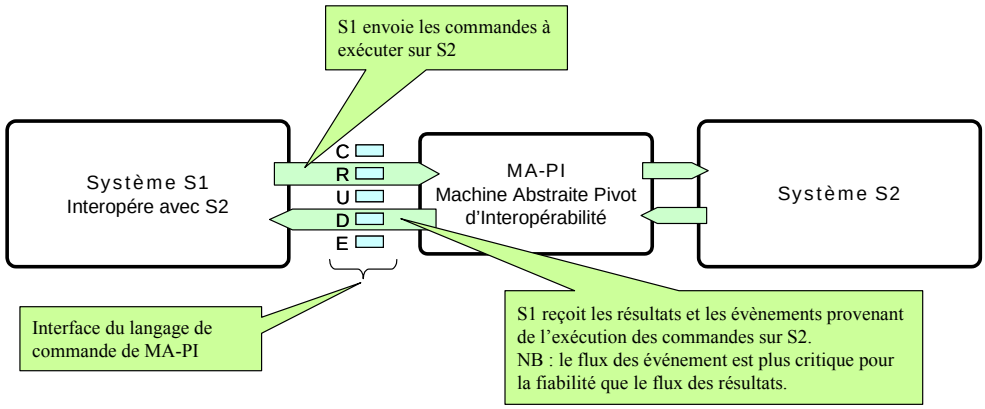


Figure 4.9 : Interface de commande du langage pivot

Le système S1, via la machine de communication MIC (non représentée sur la figure) est capable de commander l'un quelconque des systèmes de la fédération via un jeu de commandes résumé par l'acronyme CRUDE que nous avons déjà rencontré avec la machine abstraite spécialisé en gestion des données, augmenté d'une commande de mise en exécution « Execute » d'un service du système S2.

CREATE

Correspond à l'assignation d'une ressource de S2 dont le pilotage de niveau 2 (PE selon notre terminologie) est effectué par S1.

RETRIEVE

Correspond à l'obtention d'une information (peu importe le mode d'obtention, TIRE/PULL ou POUSSE/PUSH), équivalent à un ordre de lecture d'un support externe (READ) ou à la réception d'un message (RECEIVE).

UPDATE

Correspond à une opération de mise à jour du système S2, équivalent à un ordre d'écriture (WRITE) ou à l'envoi d'un message (SEND).

DELETE

Correspond à l'opération inverse de CREATE, équivalent à une libération de ressource préalablement consignée. Notons qu'une libération doit pouvoir être effectuée automatiquement après écoulement d'un délai conventionnel, analogue au TIME-OUT dans les réseaux.

Les commande CRUD concernent toutes les entités du système, indépendamment de leurs spécificités. Elles s'appliquent tout autant aux données, aux traitements (application, services, transaction selon la granularité), aux automates de contrôle (workflow).

EXECUTE

Correspond à la mise en exécution d'un service piloté au premier niveau (PI selon notre terminologie) par S2, mais dont le pilote externe PE est S1. Ce service peut être prédéfini, ou construit par l'envoi d'une suite de commande CRUD à partir de données de S1. Les commandes de la classe EXECUTE s'adressent à des entités système pour lesquelles il existe un machine abstraite capable de les exécuter (i.e. PERFORM).

Chacune des commandes renvoie un vecteur d'état conforme à la structure précisé ci-dessus (voir figure 4.6), susceptible d'interagir sur le séquencement des opérations du point de vue de S1.

L'état des opérations effectuées par S2 pour le compte de S1 est conforme aux états classiques des processus, rappelé ci-dessous.

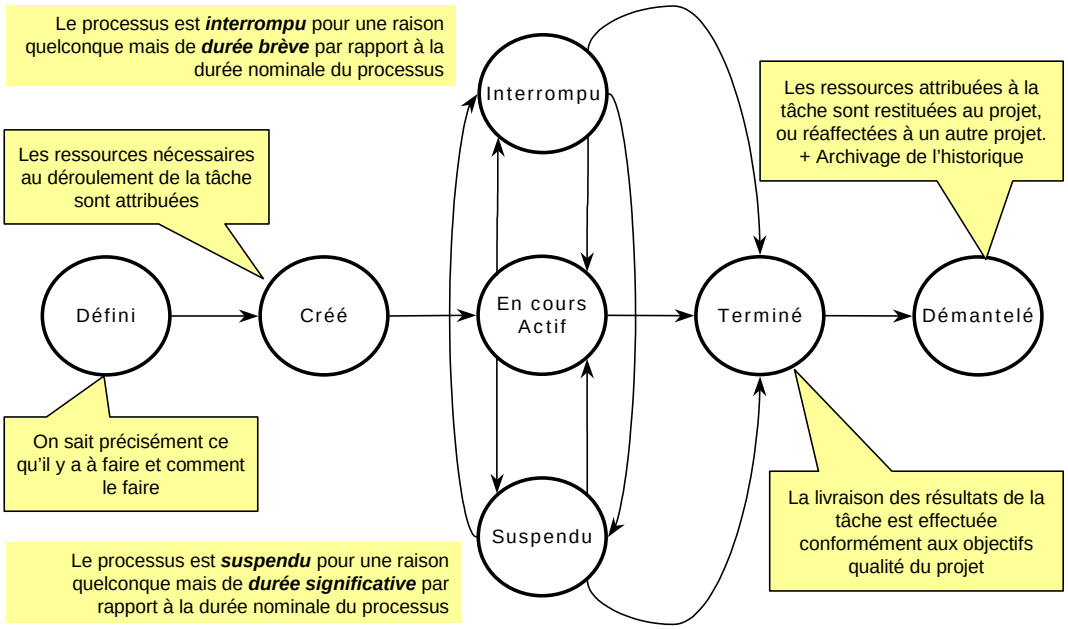


Figure 4.10 : Etats de l'exécution d'une commande

Le changement d'état du processus est transmis à S1 qui peut décider de la suite des opérations à effectuer.

Dans le cas de la commande EXECUTE, il y a effectivement exécution d'un programme construit avec le jeu d'instructions disponibles dans la machine à interopérer (voir ci-dessus). Il faut pour cela définir un langage de workflow qui doit faire partie des standards de la fédération. Ce langage ne peut être qu'un langage orienté processus, du type BPML[58], associé à un mécanisme de désignation (adressage) des entité susceptibles d'être manipulées par ce langage.

La spécification de ce langage est une étude en soi. Compte tenu de la nature très particulière de l'interopérabilité, ce langage doit être doté de tout ce qui est nécessaire à la gestion des tests d'intégration (en particulier, un support à la programmation par contrat).

Indépendamment du contenu, on peut cependant préciser la forme générale de ce langage.

Compte tenu de ce qui a été dit sur les représentations concrètes et abstraites, ce langage devra avoir les caractéristiques suivantes :

a) Représentation concrète : une forme graphique, doublée d'une forme textuelle a priori en XML pour le stockage des programmes.
b) Représentation abstraite : potentiellement une par plate-forme (voire une par organe de la machine abstraite), avec éventuellement une forme pivot correspondant à la machine abstraire PIM de la fédération de systèmes.

Pour éviter une prolifération de traducteurs, et simplifier le processus d'ingénierie de la fédération, il y a intérêt à homogénéiser les plates-formes, étant entendu qu'elles ne pourront jamais être parfaitement synchronisées. La finalité de l'architecture pivot est de faire en sorte que l'écart soit le plus faible possible, sans pour autant rigidifier l'architecture, ce qui signerait sa mort.

[58] Business Process Modelin Language ; voir le site de l'OMG.

Traçabilité et dépendances fonctionnelles

Du point de vue du cycle de développement des systèmes fédérés et de la construction des différents modèles élaborés tout au long du cycle de développement, il est essentiel que chaque étape de la modélisation des systèmes doit faire ressortir ce qui est spécifique au système (règle de subsidiarité) et ce qui relève de la fédération.

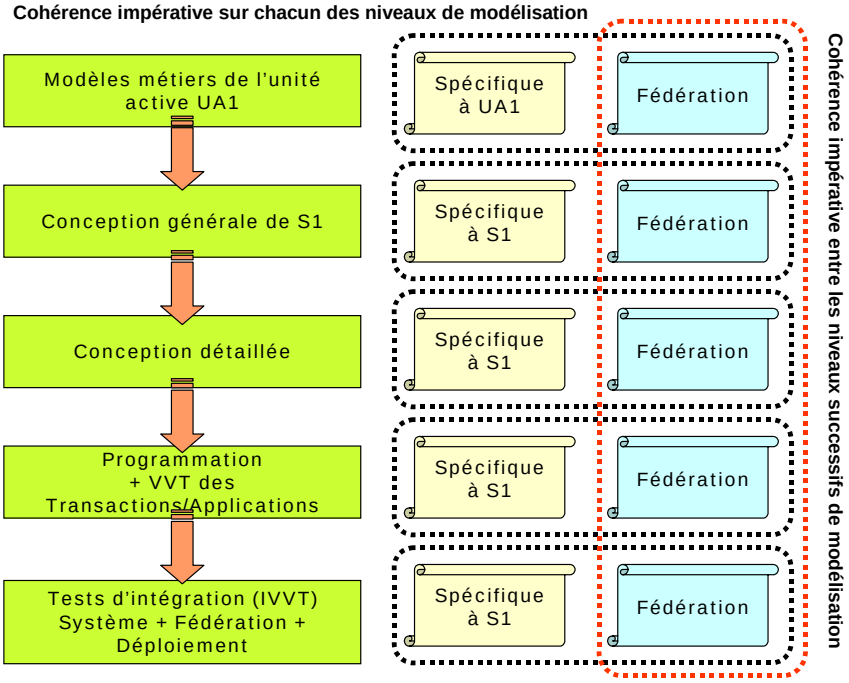


Figure 4.11 : Intégration du référentiel fédération dans les systèmes

D'un point de vue statique, les entités système et les entités fédération sont sur des cycles d'évolution a priori complètement différents.
D'une façon plus générale, le non fonctionnel (i.e. FURPSE) de ces deux catégories d'entités est très différent. Donnons deux exemples :

Exemple 1 : la fiabilité (FURPSE)

L'exigence de fiabilité est nécessairement plus forte sur une entité fédérée que sur une entité système spécifique. De par sa nature, une entité fédérées circule d'un système à l'autre. Elle est plus fréquemment utilisée, et dans des contextes d'emploi plus variés. Son potentiel de contamination est plus élevé en terme de propagation des défauts. Il est nécessaire d'avoir une trace complète de sa circulation au sein des différents systèmes.

Exemple 2 : l'évolutivité (FURPSE)

Les cycles d'évolutivité d'un système particulier et de la fédération sont totalement asynchrone. Un système particulier peut évoluer indépendamment de la fédération, tout en maintenant sa capacité fonctionnelle d'interopérabilité. Réciproquement, la fédération peut évoluer sans pour autant provoquer des mises à jour dans tous les systèmes, ce qui serait d'ailleurs en contradiction avec l'objectif recherché.
Dans les deux cas, il faut gérer les interfaces entre les entités système spécifiques et les entités fédérées, et surtout gérer tous les couplages possibles entre les deux catégories d'entités.
D'un point de vue dynamique, il faut considérer les chaînes de liaisons qui matérialisent les interactions entre les acteurs des différents systèmes S1, S2, …, Sn.
Deux points sont plus particulièrement importants :

a) L'interaction entre deux systèmes S1 et S2 se fait toujours par l'intermédiation d'une entité fédérée gérée par la machine abstraite dédiée à l'interopérabilité MA-PI (Machine Abstraite Pivot d'Interopérabilité).
b) L'état FURPSE des instances de la MA-PI dans chacun des systèmes peut être différent dans chacun des système S1 et S2.

Le schéma global de l'interaction peut être décrit comme suit :

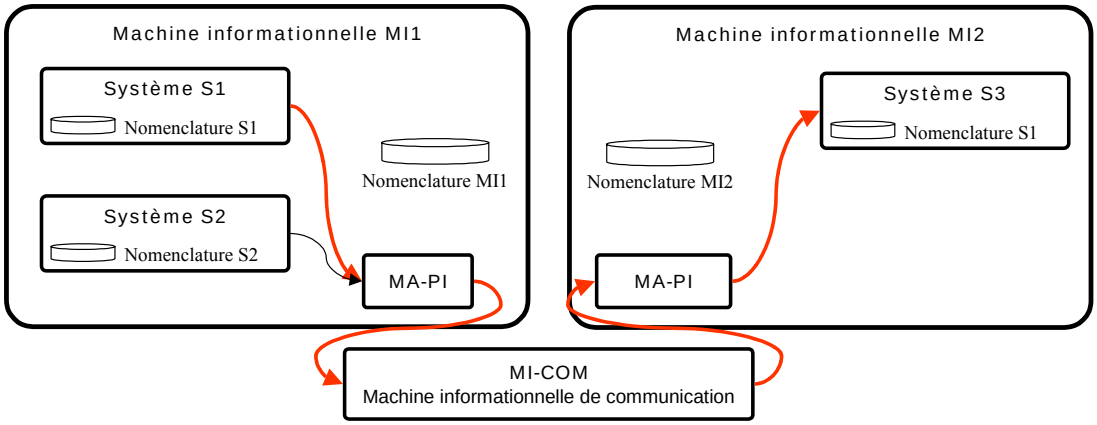


Figure 4.12 : Chaîne de liaison et machines abstraites

Les différentes nomenclatures sont des instances du concept IRDS (*Information Repository and Directory System*) introduit dans les années 80s dans la mouvance des bases de données, des dictionnaires de données et du transactionnel. Les nomenclatures doivent être cohérentes entre elles.

Une interaction complète entre différents acteurs et différentes entités peut se représenter à l'aide de diagrammes de séquences UML, selon un schéma organisé comme ci-dessous.

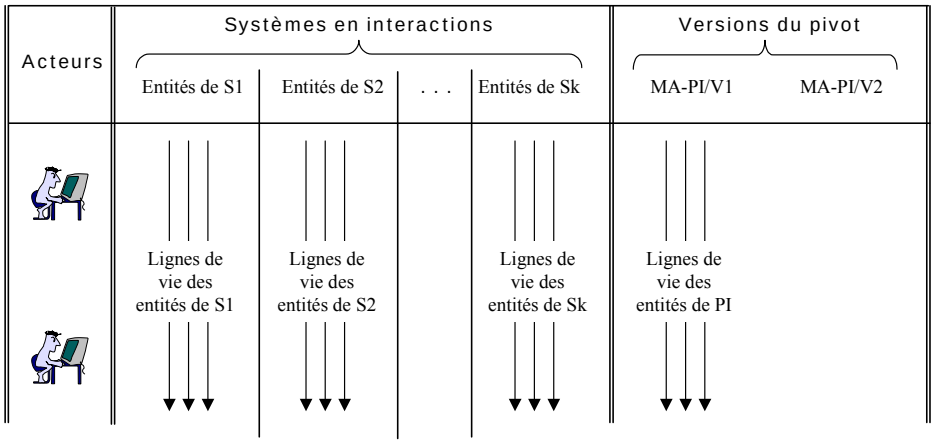


Figure 4.13 : Diagramme de séquences des interactions sur une chaîne de liaison

En fonction de la granularité des entités dont on a décidé de suivre les lignes de vie, on peut suivre les transformations successives des données associées à ces entités, et plus particulièrement lorsqu'on passe des entités systèmes aux entités MA-PI, et réciproquement.

On a vu au chapitre 2 (voir également annexe 2) l'aspect critique des couplages cachés qui peuvent complètement détruire la capacité d'interopérabilité s'ils ne sont pas complètement explicités. Les couplages concernent les données qui s'échangent, les

fonctions qui s'appellent et les évènements qui déclenchent l'exécution de tel ou tel service, ou qui signale que tel ou tel état est atteint.

La méthode fondamentale d'analyse des couplages en ingénierie système est la représentation en matrice N2[59] qui est une représentation matricielle des graphes associés ; c'est un acquis très important de l'ingénierie système qu'il est impératif de connaître.

Application aux données de l'interopérabilité

La dépendance fonctionnelle entre les données consiste à examiner de façon systématique la relation qui existe entre les données et les traitements via les chaînes fonctionnelles, et plus particulièrement celles qui impliquent les données système et les données pivot, soit :

$$\{Données\ en\ entrée\} \xrightarrow{Liste\ des\ Fonctions} \{Données\ résultat\ en\ sortie\}$$

C'est une notion très importante[60] dans la théorie des bases de données, en particulier dans l'analyse des formes normales dans le modèle relationnel. Le problème est tout à fait général.

Notons qu'il s'agit de données référencées, et non pas simplement de données figurant en paramètres dans les fonctions. En objet, cela correspond à la partie publique de l'objet.

La matrice N2 correspondante est la représentation en diagramme cartésien de cette relation. Cette représentation permet de savoir très facilement si une donnée est utilisée en lecture seule, en écriture seule, ou en mode mixte lecture/écriture. On peut également, avec une convention ad hoc distinguer si la donnée est accédé par sa référence ou par sa valeur (dans ce cas, on travaille toujours sur une copie de la donnée, ce qui est considéré comme préférable).

A l'intersection d'une ligne et d'une colonne, on donne la liste des fonctions pour lesquelles la relation est VRAI.

La taille de la matrice dépend de la granularité des données. Dans le cas de l'interopérabilité telle qu'elle nous intéresse, on pourra travailler à deux niveaux de grain :

a) Le niveau TYPE/TRANSACTION, à partir du référentiel des types constitué avec le MCD pivot,
b) Le niveau MESSAGE-FLUX/SERVICE-APPLICATION, à partir des MCD-F.

Cela montre une nouvelle fois l'importance des partitions effectuées sur les ensembles duaux $\{\text{DONNEES}\} \times \{\text{TRAITEMENTS}\}$.

La matrice $\{\text{DONNEES}\}$ à l'allure suivante :

[59] Cf. NASA *Systems engineering handbook* ; également JP.Meinadier, *Ingénierie et intégration des systèmes*, Hermès, qui donne de nombreux exemples.

[60] Pour une analyse approfondie, voir J.Ullman, *Principles of database and knowledge-base systems*, Vol. 1, Computer Science Press.

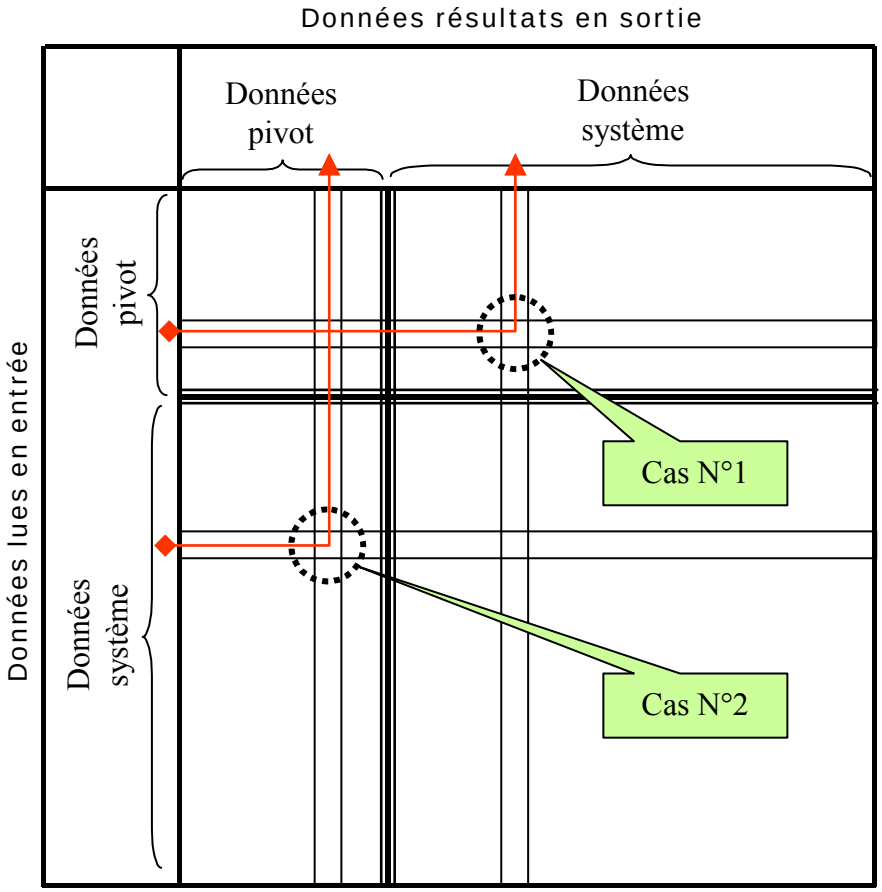


Figure 4.14 : Matrice N2 de couplage des données

Toutes les données figurent sur chacun des cotés de la matrice, en faisant apparaître les données pivot et les données système, en distinguant les données en entrée et les données résultats conformément à la relation de dépendance fonctionnelle. Les couplages qui nous intéressent sont ceux des régions latérales cas1 et cas 2.

Région cas N°1

Correspond à la situation où une donnée pivot est en mode lecture dans l'une quelconque des fonctions du système, et une donnée système est en mode écriture dans ces mêmes fonctions.

Plusieurs fonctions peuvent établir la relation de dépendance.

Si le pivot évolue, il y a potentiellement évolution concomitante des données systèmes impliquées dans cette relation. Tout dépend si le type de donnée système inclut complètement le type de la donnée pivot. En application du principe de compatibilité ascendante, via le mécanisme d'extensibilité vu ci-dessus (Figure 4.8), certaines défaillances peuvent être compensées.

Le tableau ci-dessous précise les deux situations possibles, compte tenu de l'état des pivots.

Pivot émetteur	Pivot récepteur	Données système en mise à jour	Remarques
Etat(n)	Etat(n-1) Pas de conversion disponible chez le récepteur	Diagnostic Défaut de mise à jour potentiel	Le pivot récepteur peut demander une re-émission dans l'état(n-1) nécessairement connu de l'émetteur
Etat(n-1)	Etat(n) Le pivot récepteur dispose nécessairement de la conversion	Pas de problème en mise à jour	Le pivot émet une mise en garde (warning) vers le système récepteur

Région cas N°2

Correspond à la situation où une donnée système est en mode lecture dans l'une quelconque des fonctions du système, et une donnée pivot est en mode écriture dans ces mêmes fonctions.

Si les règles qui déterminent le bon usage du pivot ont été respectées, il n'y a pas de problème car c'est au système d'être en conformité avec les règles de la fédération. La matrice N2 permet d'identifier les audits à faire pour s'en assurer, dans la conduite du projet d'interopérabilité.

Un problème se pose si le système fait évoluer ses données, car il faut alors s'assurer que des dépendance transitives (dans la région diagonale) ne finissent pas par contaminer le pivot. La matrice N2 permet d'analyser les problèmes liés à la transitivité, et de calculer les chemins correspondants (opérations classiques en théorie des graphes).

Le tableau ci-dessous précise les deux situations possibles, compte tenu de l'état des pivot.

Données système en lecture	Pivot émetteur	Pivot récepteur	Remarques
Jamais de problème en lecture	Etat(n-1)	Etat(n) Le pivot récepteur dispose nécessairement de la conversion	
	Etat(n) Le pivot émetteur dispose nécessairement de la conversion (n-1)	Etat(n-1) Le pivot récepteur peut demander une re-émission dans l'état(n-1)	Le pivot récepteur émet une mise en garde (warning) vers le système récepteur

NB : en structurant le pivot et les systèmes, par exemple avec des critères métiers, on peut faire des analyses de même type à l'intérieur du pivot, et à l'intérieur des systèmes, et appliquer les mêmes principes d'organisation (cela revient à créer de l'ordre). L'évolutivité globale n'en sera que meilleure.

Application aux fonctions (transaction, service, application) de l'interopérabilité

C'est exactement le même principe sauf que cette fois on analyse la relation {APPELANT}×{ APPELE} qui constitue le graphe de contrôle de l'ensemble {système + son pivot}.

La matrice a l'allure suivante :

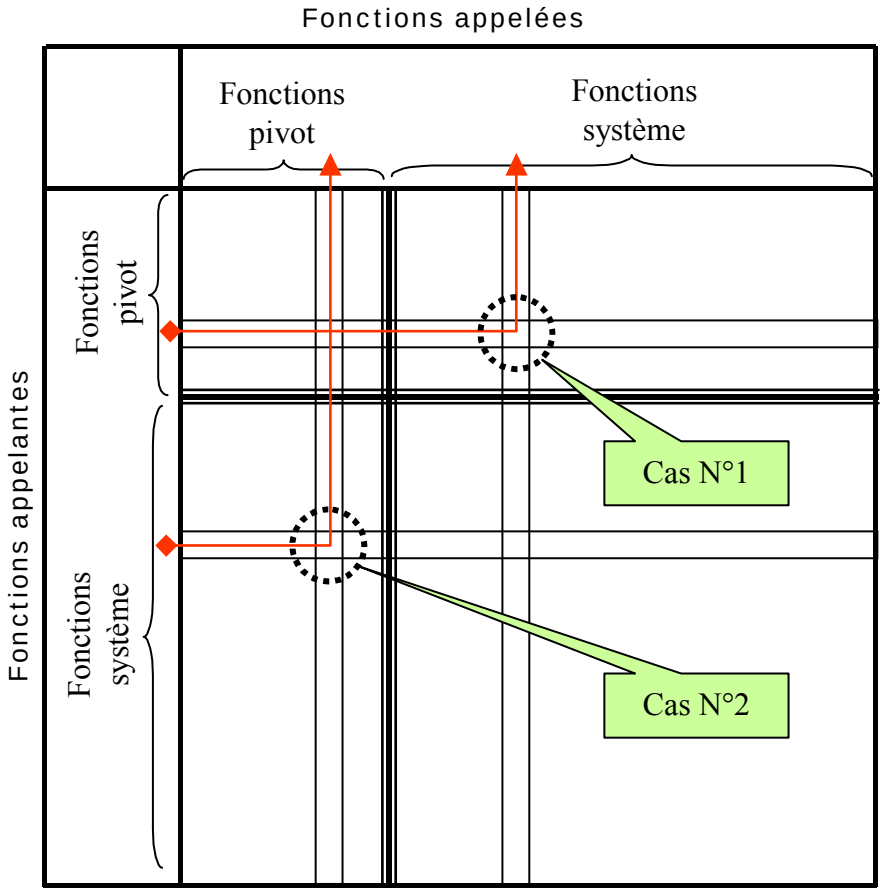


Figure 4.15 : Matrice des fonctions {APPELANT}×{ APPELE}

On lit à partir de la ligne :

La fonction Fx $\xrightarrow{Appelle}$ *La fonction* Fy

A l'intersection on peut préciser la nature de l'appel, soit séquentiel, soit asynchrone (lancement d'une activité correspondant à la fonction Fy). Dans le cas séquentiel, le retour normal se fait sur l'appelant. Dans le cas asynchrone le retour se fait sur un (ou plusieurs) point de synchronisation dont on peut préciser la localisation dans une liste de fonctions dans lesquelles le retour est susceptible de survenir.

NB : cela revient à appeler le moniteur d'événement, en précisant la classe des évènements permettant un retour à ce point précis.

La sémantique de communication inter-fonctions est très dépendante des plates-formes ; ceci doit être une incitation à faire une étude purement fonctionnelle, en privilégiant les synchronisations induites par la logique métier, ce qui permettra de choisir la plate-forme en toute connaissance de cause. Si celle-ci est une donnée de l'architecture, cela permet également de mieux mesurer la distance [conceptuel, logique/physique], très importante pour calibrer l'effort de IVVT.

Tout appel implique des manipulations de données ; on retrouve les mêmes cas1 et cas 2 que précédemment.

Application aux événements résultant de l'interopérabilité

Lorsque qu'un système S1 sur une MI souhaite faire travailler un système S3 pour son compte (voir la figure 4.12, ci-dessus), S1, ou une fonction de S1, va notifier à S3 un ensemble d'évènements déclenchant, à partir desquels seront générés un ensemble

d'événements réponses qui permettront à S1 de suivre la progression des activités exécutées (i.e. PERFORM) par S3.
La machine MA-PI associée à S1 joue alors le rôle de pilote d'événements ; c'est un moniteur qui fonctionne sur un mode STIMULUS → REPONSE.
D'un point de vue statique il est indispensable de construire une première matrice qui permet de savoir exactement, fonction par fonction, quels sont les événements émis et quels sont les évènements reçus (réponses attendues), soit :

{Ensemble des événement : e1, e2, …, en}×{Ensemble des fonctions émettrices/réceptrices d'évènements : FER1, FER2, …, FERk}

Règle :
Une première règle évidente stipule qu'aucun événement émis ne peut rester sans réponse, même si un TIME-OUT a été atteint. Il doit toujours exister une fonction réceptrice de l'événement, doublée d'un mécanisme de récupération par défaut.

L'ensemble {FER} est un sous ensemble des Fi du système S1, et on comprend tout de suite pourquoi cet ensemble doit être très petit.

NB : Dans les modèles de programmation ancien, du type FORTRAN/COBOL, cet ensemble, pour le programmeur d'application, est vide : c'est une programmation simple, permettant de travailler en évitant d'avoir à connaître les mécanismes fins de la machine.
Dans les modèle de programmation temps-réel ou transactionnel, il faut une sémantique pour gérer la concurrence ; l'ensemble n'est plus vide. Deux cas sont à considérer :

a) En transactionnel, le moniteur peut généralement régler tous les problème, moyennant un style de programmation avec des ordres COMMIT, ROLLBACK, ABORT, etc. la programmation est moins simple, mais reste gérable par des programmeurs COBOL classiques.
b) En temps réel, c'est l'application qui règle les problèmes, d'où une programmation beaucoup plus complexe, qui requiert une formation adaptée, mettant en œuvre des driver d'exceptions, avec des conflits potentiels avec la gestion de ressource du système d'exploitation sous-jacent.

Il faut donc être lucide quand il faut gérer des événements.

L'ensemble des services {Svi} que MA-PI peut piloter est par ailleurs connu.
Du point de vue du système S1, un train de travail à faire exécuter par MA-PI pourra avoir l'allure suivante :

SV1, SV2 { SV3 | SV4} SV5 (SV6 | SV7) SV8

{} signifie lancer en // ce qui est à l'intérieur, et attendre la terminaison pour continuer.
() signifie un choix entre plusieurs alternatives mentionnée à l'intérieur.
Du point de vue des événements, cela signifie :

- `Lancer` SV1
- `Attendre` la terminaison de SV1 (événement FIN NORMAL SV1)
- `Lancer` SV2
- `Attendre` la terminaison de SV2 (événement FIN NORMAL SV2)
- `Lancer` SV3
- `Lancer` SV4
- `Attendre` la terminaison de SV3 ET SV4 (événement FIN NORMAL SV3 et FIN NORMAL SV3 ; l'ordre d'occurrence est indifférent)

- `Lancer` SV5
- `Attendre` la terminaison de SV5 (événement FIN NORMAL SV5)
- `SI` vecteur d'état SV5 = telle valeur, `Lancer` SV6
- `SI` vecteur d'état SV5 = telle autre valeur, `Lancer` SV7

etc. Ceci n'est bien sûr qu'une ébauche, et on peut imaginer toutes les complications avec la gestion des terminaisons anormales.
D'un point de vue dynamique, la prise en compte des événements reçus peut dépendre de l'état du système S1. Tel événement sera accepté dans l'état E1 de S1 mais ne le sera pas dans l'état E2. Il faut donc être à même de préciser les évolutions de cette fonction d'état, ce qui conduit à spécifier une machine à état fini, i.e. un automate de workflow.
C'est en fait un plan de travail que la MA-PI/S1 communiquera en tout ou partie à la MA-PI/S3 pour mise en exécution sur la machine informationnelle MI2.
Là encore, une analyse fonctionnelle rigoureuse au niveau métier permettra de choisir l'outil de workflow le plus à même de répondre au besoin, ou de programmer directement l'automate avec les outils de monitoring disponibles sur la plate-forme.

Procédé de construction d'une architecture pivot

Pour apprécier correctement les possibilités de paramétrage d'une architecture pivot, il est essentiel de bien assimiler le procédé de construction d'un système de façon à identifier les points d'articulation sur lesquels l'adaptabilité, i.e. l'ajout et/ou le remplacement de fonctions, va pouvoir être mise en œuvre.
Les grandes lignes du procédé de construction peuvent être schématisée comme suit :

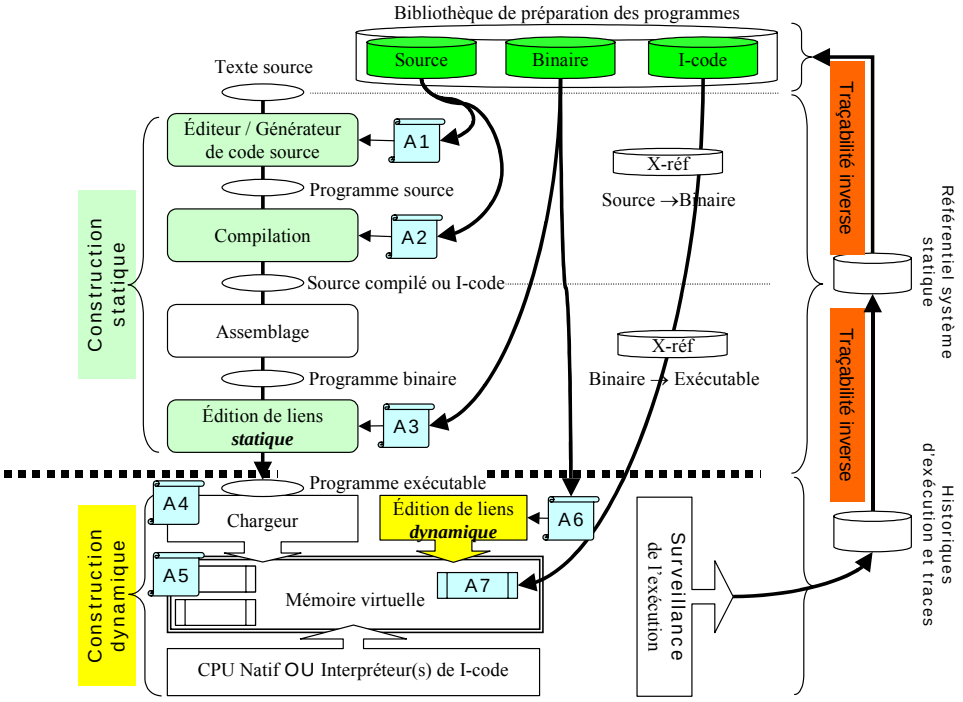


Figure 4.16 : Procédé de construction et d'intégration d'un système

Légendes :
A1 et **A2** sont des adaptations par des textes source que l'on peut incorporer au programme soit avant la compilation à l'aide d'un éditeur de texte, un pré-processeur comme dans C/C++ ou un générateur de programme, soit pendant la compilation avec les fonctions du langage prévue à cet effet (`copy`, `include`, etc.).

A3 est un mécanisme de l'édition de liens statique qui permet d'incorporer au programme des objets binaires au format des unités de compilation.
A4 est une fonction du moniteur de travaux de la plate-forme qui permet d'incorporer au programme, lors de son lancement, un texte (ce sont des données) qui pourra être lu par le programme à l'aide d'APIs système ad hoc disponibles sur la plate-forme.
A5 est tout simplement un fichier de paramétrage que l'application pourra consulter chaque fois que nécessaire (par exemple des « constantes » pas assez immuables pour pouvoir être déclarées ; 3.14159… est une vraie constante, tant qu'on est en géométrie euclidienne ; la TVA à 19,6% est une « constante » lentement variable).
Les possibilités A1 à A5 existent depuis plus de 30 ans dans les langages de programmation et les systèmes d'exploitation ; A6 et A7 sont beaucoup plus récentes dans la pratique, bien que connues également depuis plus de 30 ans, car elles nécessitent beaucoup de ressource de performance mémoire + temps CPU.
A6 est la possibilité d'incorporer dynamiquement, au moment de l'exécution du programme, du code et/ou des données, lors de leur première référence. C'est le mécanisme d'édition de lien dynamique, apparu pour la première fois avec le système MULTICS du MIT. C'est un mécanisme d'une extrême puissance, très consommateur de ressources, qui permet de différer au plus tard des références à des informations qui faute de quoi auraient du être liées statiquement via l'éditeur de liens. C'est en fait une façon de mettre à jour dynamiquement l'application, sans avoir besoin de tout recompiler, et surtout de tout ré-intégrer (mais il faut tout de même valider). Seul le protocole d'appel du programme doit être standardisé, le corps viendra au dernier moment.
A7 est la possibilité de masquer la machine réelle hardware par une machine purement logicielle, comme la JVM (*Java Virtual Machine*), ce qui permet de disposer d'autant de types de machines que de besoin, donc une très grande souplesse, au prix :

a) d'une baisse de performance très importante (facteur 100, voire plus !), ce qui avec les microprocesseurs actuels ne pose plus de problème dans certains contextes comme les IHM, et
b) d'une augmentation de complexité très importante, car le processus d'IVVT déjà problématique et difficile en mode statique, est cette fois complètement dynamique ; en conséquence, la stratégie d'intégration est intégralement à revoir.

Les machines abstraites comme celles mentionnées ci-dessus, peuvent être réalisées sous la forme de machines virtuelles comme la JVM.
La possibilité de compléter dynamiquement une application avec de nouvelles fonctions mieux adaptées au contexte, soit par édition dynamique, soit par ajout d'une machine virtuelle, est une capacité fonctionnelle particulièrement bien adaptée au contexte de l'interopérabilité, mais encore faut-il soigneusement doser les parties dynamiques de l'application pour ne pas rendre l'ingénierie de l'intégration carrément ingérable, tant au niveau de l'équipe d'intégration, que de la logistique que cette nouvelle forme d'intégration implique. C'est ce qui figure sur la partie droite du schéma avec les mécanismes de surveillance et les traçabilités inverses. Sans ces mécanismes, sur lesquels repose toute l'intégrité du dispositif d'adaptation dynamique, il est exclu de faire fonctionner correctement une telle architecture et de donner une quelconque garantie de qualité.
Pour qu'une machine virtuelle inspire le même niveau de confiance qu'une machine réelle à laquelle elle prétend se substituer, elle doit avoir le même niveau de maturité, donc avoir été exposé dans de nombreux contextes différents, et pendant longtemps.

C'est le cas des « moteurs » relationnels actuels qui utilisent couramment une partie de ce type de facilités, par exemple la possibilité de fabriquer dynamiquement des instructions SQL en format caractères, de les compiler et de les exécuter dans la foulée (NB : il a fallu quelques années avant que ça ne marche parfaitement !).
Les fichiers I-code, figurant sur le schéma, contiennent les codes des programmes destinés à telle ou telle machine virtuelle (ce peut être du code binaire, ou des formats caractères selon les caractéristiques de la performance souhaitée).

Assemblage et intégration des entités de l'interopérabilité

Trois catégories d'entités interviennent dans les assemblages nécessaires à la construction de l'architecture pivot :

a) Les données,
b) Les traitements,
c) Les contrôles.

Examinons chacune des catégories avec comme point de vue la logique d'intégration. On a vu précédemment qu'il était commode, sinon nécessaire, de faire apparaître trois niveaux d'abstraction des entités constitutives de l'architecture pivot, soit en allant du plus général au plus spécifique : le niveau fédération, le niveau système, et le niveau sous-système où apparaissent les entités informatiques concrètes comme les applications et/ou les transactions.

Assemblage et intégration des données

Concernant les données, on a l'habitude de distinguer (depuis le langage PASCAL, de N.Wirth) le type de la donnée de son instance en mémoire[61]. Autrement dit on désolidarise la sémantique portée par le type de la représentation en mémoire qui dépend à la fois du support hardware et de l'implémentation faite par le compilateur. Ceci permet de raisonner sur le type indépendamment de la diversité des instances : le type est donc un moyen descriptif puissant qui permet de simplifier la modélisation des problèmes, car il permet de passer du particulier (i.e. les individus) au général (i.e. les classes).
On distingue également les types primitifs (et les données scalaires que ces types permettent de construire) et les types agrégés qui correspondent à des entités de plus haut niveau. Là encore, le langage PASCAL a été un tournant décisif. Les données agrégés ont été bien sûr présentes dès les premiers langages de haut niveau comme COBOL pour décrire le contenu des fichier à l'aide des concepts de `record`, puis de schémas des bases de données du modèle réseau, dès le début des années 70s.
D'un point de vue syntaxique, types et données scalaires constituent un lexique avec lequel on va pouvoir engendrer, à l'aide d'une grammaire, des types et des données agrégés de plus haut niveau comme les fichiers mono ou multi-types, les MCD, des schémas de bases de données. Ces deux niveaux de description font partie, depuis ALGOL 60, du standard de présentation de tout langage de programmation, du moins lorsque l'on s'astreint à une certaine rigueur.
D'autres relations que l'agrégation sont nécessaires pour décrire les données, en particulier pour ce qui concerne les contraintes d'intégrité qui sont des conditions nécessaires au sens des données liées par ces contraintes (cf. les contraintes pour les formes normales du modèle relationnel, et plus généralement ce qui est visé par des langages comme l'OCL de la norme UML, qui est une invention d'IBM antérieure à

[61] En physique, il y a longtemps que l'on distingue le type d'une grandeur, i.e. son équation de dimension, de sa représentation dans un système d'unité particulier ; les équations de dimensions permettent d'étudier la forme des lois physiques, uniquement sur des considérations qualitatives.

UML). En application de la règle d'Occam, on peut d'ailleurs s'interroger sur la nécessité d'un langage comme OCL (ou de tout autre du même type) par rapport aux langages de programmation classiques, car tout ce qui est exprimable en OCL l'est également en JAVA, C++, … ? Où est la simplification conceptuelle dont se réclame les dévots de l'objet ? Certainement pas en multipliant les langages !
La distinction entre le niveau lexical et le niveau syntaxique est doublement utile, d'abord pour les acteurs métiers qui retrouvent des catégories linguistiques avec lesquelles ils sont nécessairement familiers, et ensuite pour les architectes et réalisateurs qui vont pouvoir les exploiter pour diverses opérations de paramétrages concernant chacun des niveaux.
Cette présentation de la structure des entités décrivant les données va être utilisée systématiquement sur les trois niveaux d'abstraction qui nous intéressent pour modéliser l'interopérabilité, comme le montre le schéma ci-dessous.

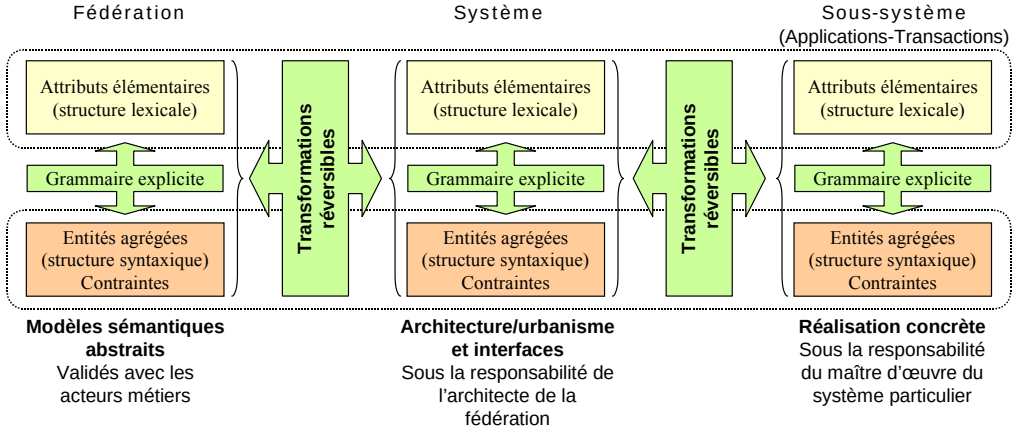


Figure 4.17 : Représentation des trois niveaux d'abstraction

Ce modèle de transformation généralise de façon systématique les ébauches proposées au chapitre 2, *Problème des données fédérées*. Chaque niveau d'abstraction doit être syntaxiquement cohérent, et spécifié dans un formalisme grammatical permettant l'analyse et la génération sans ambiguïté des textes correspondants. La grammaire correspondante doit être « *contexte-free* », c'est à dire que toute construction dont la validité fait intervenir des considérations sémantiques est exclue ; c'est une condition nécessaire à l'obtention de traducteurs simples et structurables.
NB : L'expérience montre qu'il est très facile, pour un non spécialiste, de spécifier des langages non *context-free*, difficiles à traduire d'une part, et à l'usage très difficile à faire évoluer, voire même à mémoriser par les usagers car rien n'est plus arbitraire que la sémantique. Le langage humain est très fortement dépendant de la sémantique, ce dont à horreur l'informatique.
Les règles de traduction d'un niveau à l'autre, sont des règles de transformations (i.e. un système de re-écriture, plus ou moins complexe, piloté par des automates de traduction fondés sur les grammaires des langages) dont la simplicité dépend essentiellement de la nature non contextuelle du langage source et du langage cible qui doivent donc être définis avec la plus grande rigueur.

Le niveau d'abstraction correspondant à la fédération est le plus général. Il doit être indépendant des différents systèmes qui constituent la fédération, et bien sûr indépendant de toute considération relevant des plates-formes informatiques. Ce niveau est purement conceptuel (ontologique, pour parler comme les partisans du tout objet), c'est à dire que les entités et attributs propres à ce niveau ne s'intéressent qu'au sens

brut, indépendamment de toute spécialisation liée au traitement, par exemple : distance, altitude, azimut (c'est un angle), surface-zone (donc également un contour), forme géométrique, texte structuré comme on en manipule à travers tous les formulaires de notre vie courante, des agrégats (vecteur, matrice, arborescence, etc.), des ensembles, des individus particuliers, des classes d'individus (il existe une relation d'appartenance), etc.
A ce stade de la modélisation, le modèle est essentiellement un système a) de noms, auxquels sont attachés des propriétés générales, b) de règles de structures comme celle du modèle ENTITE – RELATION – ATTRIBUT (ERA) permettant de préciser l'organisation des agrégats, et c) de classification par thèmes sémantiques conformément aux point de vue des acteurs concernés par ce niveau. La sémantique est exprimée en langage naturel, augmenté de représentations graphiques, car in fine c'est le seul langage commun à tous les acteurs.
Les types génériques, les MCD de flux (i.e. le contenu, en tant que sémantique véhiculée), les MCD métiers font partie de ce niveau. Le formalisme fondamental est le modèle ERA (dans le jargon MDA, c'est un méta-modèle qui sert à décrire les MCD) dont le fondement est la théorie des ensembles, la logique des classes et des relations, ce qui est évidemment bien plus important que le qualificatif méta qui est purement tautologique.

Au niveau système, on procède à une première étape de spécialisation. Un système particulier peut avoir sa propre logique de modélisation, mais nous avons vu qu'elle ne doit pas contredire la logique de la fédération auquel le système doit nécessairement s'adapter pour précisément s'intégrer à la fédération. Ce niveau est le plus délicat car il doit préparer le travail de réalisation, en parfaite conformité avec la logique métier qui est celle du niveau fédération. C'est le niveau où le talent conceptuel et l'expérience des architectes modélisateurs va se révéler décisif. C'est là que sont créées véritablement les abstractions intermédiaires (cf. figure 3.4), comme les applications, les transactions, les services qui sont des regroupements dont les critères sont multiples : métier, technique, technologique, abstractions devant prendre en compte différentes caractéristiques non fonctionnelles qui ont une visibilité au niveau système.
Ce qui n'était qu'un grandeur numérique au niveau fédération peut se spécialiser en nombre réel (au final, ce sera un nombre flottant, qui comme chacun sait ne sont pas vraiment des nombre réels !), en nombre décimal (avec une logique d'arrondi), ou en nombre entier. A ce stade on peut ne pas encore avoir besoin de notions concernant l'ordre de grandeur, la précision, l'intervalle de définition qui seront indispensables à la réalisation. Un vecteur à deux dimensions peut être représenté par un nombre complexe.
En matière de grandeurs, c'est généralement au niveau système que l'on précisera si l'on parle en système métrique ou en unités anglaises, en Euros ou en Dollars ; que l'on précisera le système spatio-temporel utilisé et les modes de représentation des entités géométriques ; que l'on caractérisera les ensembles, soit par leur extension (tous les éléments qui les constituent sont connus), soit par leur compréhension (on sait vérifier l'appartenance à l'aide d'une fonction ad hoc).
En matière de modèles de données, on peut spécialiser les MCD du niveau fédération sur des modèles de données déjà connus comme les modèles hiérarchiques, réseaux, relationnels, objets, voir un modèle ad hoc pour représenter des entités particulières (information géographique, trajectoire dans un référentiel spatio-temporel, etc.). A ce stade, il est encore légitime de ne pas se lier à une technologie particulière, quand bien même cette technologie implémente l'un des modèles retenu pour le système.

Au niveau sous-système, on aura complète liberté pour utiliser toutes les ressources descriptives des technologies disponibles sur la ou les plate(s)-forme(s) de réalisation, dans le respects des règles. On peut continuer à spécialiser, mais sans contredire ni créer d'ambiguïté sur les objets échangeables au sein de la fédération, de façon à préserver la réversibilité des traductions. On peut également choisir d'encapsuler à ce niveau toutes les dépendances vis à vis des technologies conjoncturelles sous-jacentes, en définissant des concepts d'appareils virtuels comme cela est fait dans les systèmes d'exploitation vis à vis des dispositifs hardware.

Assemblage et intégration des traitements

Nous avons insisté à différentes reprise sur l'importance fondamentale du concept de transaction, en faisant ressortir le fait qu'une transaction est une opération élémentaire qui doit avoir un sens du point de vue métier. Par exemple, un opération DEBIT – CREDIT a un sens pour un banquier ; l'affectation d'une partie des stocks de carburant à telle ou telle unité active a un sens pour un logisticien responsable des approvisionnements.

La transaction est un quantum sémantique élémentaire, associée au modèle de données sur lequel elle agit : les transactions sont les opérations élémentaires (i.e. atomiques, mais la taille de l'atome est une décision de l'architecte) du modèle de données du niveau auquel elles se rattachent. Ce sont donc des unités de sens à partir desquelles on va pouvoir, par agrégation, construire des services plus élaborés. On appellera SERVICE un ensemble d'opérations effectuées dans un certain ordre qui a de la valeur pour un acteur du système (métier, support, autre système). La distinction transaction – service est quelque peu arbitraire, car un service peut ne comporter qu'une seule transaction. Cependant, des considérations d'ordonnancement doivent stipuler ce qui forme un bloc insécable, par définition non interruptible, de ce qui peut être interrompu et repris un peu plus tard. Les projets humains font couramment cette distinction qui ne posent aucun problème du point de vue métier (NB : du point de vue informatique, c'est une autre paire de manche !). Un service sera par définition interruptible, ce qui n'aura d'intérêt que si il est formé d'au moins deux transactions.

On retrouve pour les traitements la même structuration en niveau lexical et syntaxique, la structure lexicale étant formée des opérations élémentaires autorisées par le modèle, la structure syntaxique définissant des opérations agrégées, de plus haut niveau, et ce jusqu'à l'obtention de services (NB : dans un langage de programmation, c'est la distinction entre les opérateurs et les instructions).

D'un strict point de vue sémantique, une opération est définie de la façon suivante :

- Le nom de l'opération, c'est à dire ce qui porte le sens général de l'opération.
- Les données (i.e. les types) acceptables en entrée.
- Les données (i.e. les types) acceptables en sortie.
- Le contexte auquel l'opération peut se référer.
- Le vecteur d'état VE qui décrit qualitativement le résultat obtenu et l'historique de la transformation opérée (NB : on a vu, ci-dessus, qu'il était impossible de le représenter par un simple type énuméré).
- Les ressources nécessaires au bon déroulement de l'opération.
- Les contraintes d'intégrité, de surveillance, …, bref, tout ce qui définit le « contrat » que doit satisfaire l'opération du point de vue de son environnement (cf. notre modèle VEST, ci-dessus).

La définition des services nécessite la spécification d'une grammaire qui, à partir des opérations élémentaires, va permettre de construire les enchaînements des opérations constitutives du service (NB : on peut prendre les règles d'enchaînements courantes des

langages de programmation, comme celles de Java ; ou des représentations plus graphiques comme des diagrammes d'activités, ou des diagrammes états – transitions du langage UML).
Un service est défini de la façon suivante :

- Le nom du service, c'est à dire la valeur apporté par ce service.
- La liste des opérations élémentaires constitutives du service.
- La description des enchaînements (i.e. le programme réalisant le service).
- Le vecteur d'état du service qui est une synthèse des vecteurs d'états VE des opérations élémentaires.

Comme indiqué précédemment, un service peut être interrompu par la survenu d'un événement qui rend la poursuite de son exécution impossible, ou sans intérêt du point de vue valeur.
L'ensemble {Opérations/Transactions, Services} constitue une application. Le regroupement en application est de nature purement conventionnelle, selon des critères sémantiques métiers, techniques ou technologiques, selon les niveaux de modélisation concernés.
Comme pour les données, opérations/transactions et services vont se spécialiser, conformément aux dispositifs qui existent au niveau système et au niveau sous-système. On a vu précédemment que les machines abstraites étaient un puissant moyen de définir rigoureusement les capacités de traitement de ces deux niveaux, en dosant la spécialisation de certaines d'entre elles (MA de gestion de données, MA de communications, etc.).
La construction des machines abstraites est évidemment tributaire des plates-formes choisies pour les réaliser. Nous avons à cet effet défini un concept de machine informationnelle qui est en fait un gestionnaire de ressources pour le compte des machines abstraites que la machine informationnelle héberge, ce qui fournit un autre niveau de paramétrage des plates-formes. L'ensemble de ces machines définit de modèle d'exécution de la machine informationnelle, sur lequel vont s'appuyer les machines abstraites orientées métiers ou orientées interopérabilité. Notons que tout cela revient à remonter le concept de système d'exploitation (gestionnaire des ressources hardware pour le compte des programmes), au niveau d'une gestion de ressources nécessitée par les applicatifs métiers.
A ce stade de la modélisation, nous avons défini en ensemble de MA emboîtées les unes dans les autres, avec des règles de constructions définies par des grammaires et une sémantique qui est celle des instructions d'un niveau de modélisation déterminé. On peut alors envisager des traductions d'un niveau à l'autre qui pourront mettre en œuvre toutes les ressources des techniques de compilation telles qu'on les pratique dans les compilateur ou les SGBD. Le pré-requis est bien sûr de connaître ces techniques.
Au stade ultime de la spécialisation, au niveau des sous-systèmes, voir plus bas si l'on a décidé d'encapsuler tout ou partie des technologies, on aura affaire aux modèles d'exécution et de programmation des différentes plates-formes.

En matière de traduction de modèle, il y a une règle d'ingénierie non écrite, mais fondamentale, qui stipule que pour traduire un modèle M1 dans un modèle M2, et exploiter correctement les ressources de M2 de façon automatique, tout ce qui existe dans M2 doit déjà exister, sous une autre forme, dans le modèle M1. Si ce n'est pas la cas, cela veut dire que pour tirer partie des possibilités de M2, il faut donner à l'usager du modèle M2 la possibilité d'intervenir sous une forme à définir dans le texte généré à partir de M1.

NB : ce type de dispositif existe dans certains langages de programmation (cf. le concept de `pragma` dans Ada), ou dans les langages de certains SGBD (cf. le concept de `trigger`, ou de `data-base procedure`).
Ce type de chirurgie est évidemment extrêmement problématique, et éminemment dangereuse. De deux choses l'une :

a) Soit on laisse l'usager libre de ses actes et responsables de ses actions, et on lui donne la possibilité d'intervenir dans le texte généré, ce qui présuppose qu'il maîtrise la logique de génération et la sémantique fine de M2. Ceci est parfaitement contradictoire avec une logique de réutilisation, et parfaitement illusoire entre le niveau fédération (ce sont des acteurs métiers, non informaticiens) et le niveau système. De fait, les programmes SQL qui utilisent la notion de trigger sont non portables, quand bien même il s'agisse du modèle relationnel. Dans le cas de l'interopérabilité, cette « facilité » doit être totalement exclue. En cas de défaillance, où est l'erreur ?!
b) Soit on contraint l'usager à n'utiliser cette facilité que sous le contrôle strict d'interfaces ad hoc. Ce qui veut dire qu'il doit exister, dans le texte généré des « prises » à partir desquelles l'usager de M2 pourra insérer ses propres dispositifs sans risquer d'interférer avec la logique de génération de M1, qu'en tout état de cause il doit ignorer. C'est un problème classique d'ingénierie des système d'exploitation, pour lequel il n'y a pas de solution générale, car par définition M2 ignore le détail de M1, et on ne sait pas sur quelle base il faudrait générer des point d'insertion, là plutôt qu'ailleurs. Cette facilité ne peut concerner que la traduction du niveau système dans le niveau sous-système ; elle devra faire l'objet d'un traitement adapté dans la phase d'IVVT de la fédération.

Pour ce qui concerne l'interopérabilité, le seul recours est la logique métier, à partir de laquelle on peut envisager certaines actions complémentaires, propres à la logique informatique, que l'on laisse ouvert jusqu'au stade de la réalisation où l'on pourra définitivement « bouchonner » le service correspondant. Là encore, tout cela nécessite une très grande maîtrise des techniques de compilation et de traduction, et bien sûr de l'IVVT.

Assemblage et intégration des contrôles

Le contrôle concerne tout ce qui touche à l'ordonnancement des opérations effectuées, et ceci sur les trois niveaux d'abstraction qui nous intéressent. Il s'agit de répertorier, niveau par niveau l'ensemble des évènements qui peuvent survenir et altérer l'enchaînement des opérations.
Compte tenu des aléas rencontrés à tous les niveaux, il faut également définir un mécanisme de synchronisation, comme il en existe dans tous les projets.
Ebauchons très succinctement ce qu'il y aurait lieu de faire, sur les opérations et les services définis précédemment.
Par opération :

- Liste des conditions autorisant le lancement.
- Liste des conditions de terminaison (en particulier, exploitation du vecteur d'état VE associé à l'opération).
- Conditions d'attribution et de restitutions des ressources nécessaires à l'opération (NB : la gestion peut être statique ou dynamique).

Par service :

- Liste des conditions autorisant le lancement du service.
- Liste des conditions autorisant la suspension ou l'interruption du service.

- Liste des conditions autorisant la reprise du service après interruption ou suspension.

Pour le parallélisme et la synchronisation, on peut considérer que l'on a une hiérarchie Application (ou Tâche) → Services → Transaction/Opérations.
La disponibilité d'une ressource peut être établie à partir d'une autre ressource appelée sémaphore, en référence à ce que font les marins, ou les cheminots, pour gérer les mouvements des convois qui les concernent, avec les règles de priorités que tout le monde connaît. Ces règles font bien évidemment partie du métier.
Dans une modélisation par processus métier, au plus haut niveau, les événements correspondants concernent les niveaux de pilotage mentionnés précédemment.
A ces événements, on peut associer un jeu de messages ad hoc, et les flux correspondants, auxquels on peut attribuer certaines ressources bien particulières dans la mesure où ces flux ont des niveaux de criticité bien plus élevés qu'un simple échange d'information, car ils concernent l'intégrité des services effectués par les systèmes. Là encore, tout ceci existe au niveau métier, où tout le monde sait ce qu'est un message prioritaire, une alerte, un accusé de réception, etc.

Construction statique et construction dynamique de l'interopérabilité

A ce stade de l'étude, nous sommes capables de classer les opérations qui concernent l'évolution de l'interopérabilité (évolution des différents métiers fédérés) par rapport aux capacités d'évolution de l'informatique telles que décrites dans la figure 4.16.
L'enchaînement des transformations peut se schématiser comme suit :

- Mise à jour et évolution du Modèle d'entreprise (Business Process) de la fédération ; Impact des évolutions du métier sur les modèles informatiques → Figure 2.4.
- Evolution de chacun des modèles et caractérisation des sous-ensembles participant à la fédération → Figure 4.11.
- Intégration des évolutions dans les MA et validation des chaînes de liaisons → Figure 4.12 + les Figures 4.8 et 4.9.
- Modalité des paramétrages de la MA-PI → Figure 4.2 et Figure 4.5.
- Construction et Intégration de la PSM → Figure 4.16.
- IVVT de l'évolution effectuée conformément à la méthodologie de conduite du projet d'interopérabilité → Figure 3.7 (à partir du schéma 2.4).
- Mise à disposition des nouvelles fonctionnalités aux usagers de la fédération.

Par ailleurs, tant en ce qui concerne les types de données que les services (ce sont, comme nous l'avons déjà dit, des structures duales) nous avons fait apparaître de façon systématique :

- Un niveau générique stable correspondant à des opérations et des types primitifs.
- Un niveau programmé adaptable permettant de construire de nouveaux types et de nouvelles opérations (appelées services) sur les entités du niveau générique. On peut considérer que ce type de mécanisme a été complètement validé par les L4G des années 80s.

Concernant le procédé de constructions d'une évolution de l'interopérabilité, les choix concernent les trois mécanismes de la figure 4.16 :

- Construction statique, nécessitant une intégration complète de MA-PI (et éventuellement des différents systèmes) via les interfaces A1, A2, A3,
- Construction par édition de liens dynamiques via l'interface A6,

- Extensions autorisées par les mécanismes de programmation de la machine virtuelle considérée via l'interface A7.

En théorie, on pourrait tout ramener sur l'interface A7, mais au prix d'une consommation de ressources très importante puisque tout serait fait dynamiquement, et d'une baisse de performance concomitante dont il faudrait évaluer l'impact.

Du point de vue de son langage externe, la machine MA-PI se présente comme un texte structuré comportant :

a) Une description de types, en distinguant les types primitifs, et les types dérivés construit sur les types primitifs.
b) Une description de différents types d'agrégats correspondants aux données de la MA-PI, comme par exemple les MCD d'échanges (cf. les modèles correspondant aux flux du modèle des processus métiers), les MCD des données métiers, les MCD utilisés pour la présentation et les IHM (MCD-U).
c) Une description des transactions et des services, conformément à la logique exprimée dans la figure 4.5, où là encore on distingue les deux niveauxniveau générique N0 et niveau dérivé N1).
d) Une nomenclature d'évènements associés aux opérations et aux services.
e) Un ensemble de contraintes définissant les règles d'intégrité, la surveillance et l'administration de la machine.

Toute cette description est exprimable par l'intermédiaire d'un langage ad hoc, lui-même pourvu de mécanisme d'extensibilité permettant de générer les entités dérivées à partir des entités primitives.

Un choix de langage possible pour toutes ces descriptions est XML (utilisé comme métalangage), l'ensemble constituant le langage externe LE de la MA-PI. Ceci est cependant insuffisant pour décrire une machine car il faut un modèle d'exécution (sémantique opérationnelle, et en particulier une sémantique de traitement des exceptions).

> NB : Rappelons, pour mémoire qu'une des toute première utilisation du langage APL (inventé par K.Iverson chez IBM) avait été la description de l'interface hardware software des machines de la série IBM 360 ; APL jouant dans ce cas le rôle d'un méta-langage, avec une sémantique opérationnelle qui était celle d'APL.

Ceci ne préjuge pas de l'existence d'un langage interne de la machine MA-PI (en quelque sorte son « assembleur » de concepts sémantiques représentés par les opérations) auquel serait associé un jeu d'API MI par MI (ou plate-forme par plate-forme).

Au niveau de l'interface CRUDE (cf. figure 4.9) entre un système et la machine MA-PI tout est message. On peut distinguer les catégories suivantes :

- Messages véhiculant des données (interface CRUDe)
- Messages véhiculant des commandes qui sont des ordres à exécuter (interface crudE).
- Messages véhiculant des automates de contrôle, qui en fait sont des données particulières, puisque exécutables.
- Messages véhiculant des évènements.

Si l'architecte a décidé de doter la machine d'un langage interne LI, les dialogues possibles entre un système S et une MA-PI peuvent se faire soit via un langage externe LE, soit via le langage interne LI. On peut imaginer qu'un certain nombre de situations répétitives et prévisibles puissent être compilées dans le langage LI, ce qui économisera les ressources en cours d'exécution ; dans ce cas, le système S n'a besoin que de connaître l'API de lancement du service correspondant.

On peut imaginer qu'une situation inopinée nécessite une programmation de nouveaux types dérivés et de nouveaux services ; dans ce cas, le concepteur de l'architecture pivot peut proposer deux mécanismes :

1) Un mécanisme entièrement textuel, conforme à la syntaxe et à la sémantique du langage LE. C'est ce texte qui est envoyé à la machine MA-PI qui se chargera de le compiler et de l'exécuter.
   NB : Notons que ce mécanisme purement textuel peut constituer une machine par défaut, ce qui serait l'analogue du jeu d'instructions d'une machine réduite aux seules opérations booléennes, à partir desquelles toutes les autres sont exprimables. Ceci correspondrait à des langages comme SNOBOL ou les macrogénérateurs des années 70, ou plus récemment à des langages comme PERL ou PYTON.
2) Un mécanisme qui repose sur le langage interne LI. Le système S doit avoir une instance du compilateur LE→LI.

Dans le premier mécanisme, le traducteur LE→LI est encapsulé dans la machine MA-PI. Seule l'interface externe textuel LE est connue des différents systèmes S.
Dans le second mécanisme, le traducteur est dupliqué dans tous les S. La préparation des programmes LI incombe au système S (subsidiarité), ce qui est certainement plus souple d'emploi, mais également plus complexe, car cela entraînera très certainement des propagations incontrôlables de l'interfaces LI (d'où une assurance qualité très difficile à mettre en œuvre).
Du point de vue FURPSE, le premier mécanisme est certainement préférable, mais il faut en évaluer l'impact en terme de performance (caractéristique P).
C'est ce genre de mécanisme qui est utilisé par les systèmes d'exploitation, et les protocoles de communication, pour s'adapter à la diversité des interfaces hardware (cf. par exemple, la notion de programme et de langage canal utilisé dans les contrôleurs d'entrées-sorties[62]). Les résultats de compilation peuvent être stockée dans MA-PI, pour réutilisation éventuelle. On peut aménager le schéma 4.9 comme suit :

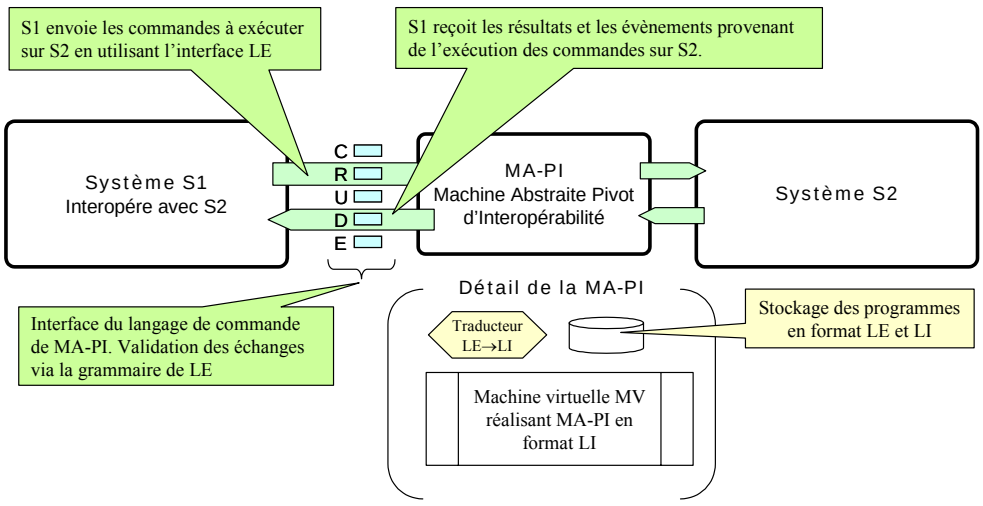


Figure 4.18 – Machine MA-PI et son traducteur

Dans une telle logique, rien n'empêche d'utiliser le traducteur LE→LI comme un service à disposition des systèmes S. Cela permet de dissocier l'opération de traduction,

62 Voir les deux livres de Henessy, Patterson, *Computer architecture* et *Computer organization and design,* Morgan Kaufmann.

de la mise en exécution, ce qui peut être une option très intéressante pour les administrateurs des systèmes qui auraient à programmer de façon inopinée (mais en respectant un minimum de règles d'ingénierie) de nouveaux services.

Toute cette problématique est au cœur de la définition et de la mise en oeuvre des Web-services[63], qui eux se situent sur un plan encore plus général (donc a priori encore plus difficile à implémenter). On peut utiliser les spécificités de la problématique d'interopérabilité axée sur les échanges pour simplifier les mécanismes, ce qui fait baisser le niveau de risque.

5. CONCLUSION

L'interopérabilité, telle qu'elle vient d'être exposée, montre que :

- L'utilisation systématique de modèles est un moyen fondamental pour clarifier et atteindre l'objectif recherché. La modélisation porte tout autant sur les contenants (syntaxe) que sur les contenus (sémantique).
- La notion de machine informationnelle, adossé sur le concept de machine abstraite, est un moyen pragmatique et concret pour articuler et organiser les relations entre la syntaxe et la sémantique des modèles. Les langages de modélisation doivent être utilisée avec cette finalité ; ils doivent être au service des modèles, et non l'inverse.
- La traduction des modèles les uns dans les autres est omniprésente à toutes les étapes de la modélisation, pour réaliser ce que nous avons appelé des transducteurs, et répercuter de façon automatique, sans intervention humaine, les modifications apportées aux modèles qui définissent les règles de l'interopérabilité.

La problématique des MDA devrait donc y trouver une place de choix, si elle arrive à surmonter sa crise d'adolescence, et à présenter les concepts qui la fondent de façon simple.

- Il est fondamental de ne pas confondre la capacité de modélisation de nature essentiellement sémantique, c'est du pur savoir-faire, et la capacité à jongler avec un langage de modélisation, UML ou autre.

L'interopérabilité est un domaine ou la complexité règne en maîtresse absolue. Quasiment tous les projets d'interopérabilité ont connu, ou connaissent, de grandes difficultés à contrôler leurs paramètres coût, qualité, fonctionnalité, délai. La mise en œuvre de middleware d'EAI n'a rien arrangé, voire même augmenté encore la confusion s'il en était besoin, et ce d'autant plus que les vendeurs de technologies ont fait croire que la complexité allait s'évanouir, comme par miracle. Les projets d'interopérabilité sont nécessairement complexes. Là encore, l'approche MDA devrait y trouver sa place, en clarifiant et en précisant les rôles des différents acteurs MOA, MOE, Architectes, etc. par rapport à ce qu'ils doivent produire en tant que modèles ou d'éléments de modèles.

Il serait naïf de croire que tout cela n'est qu'une question de langages et de méthodes. C'est là le risque majeur pour l'approche MDA, comme d'ailleurs pour UML.

---

63 cf. la littérature Web-services, en particulier, J-M Chauvet, *Services Web avec SOAP, WSDL,UDDI, ebXML*…, Eyrolles 2002, qui est une bonne introduction.

Comprendre en profondeur la problématique langages méta-langages requiert une solide formation en logique, en théorie des langages et en techniques de compilation. C'est bien autre chose que de saupoudrer son discours de méta-machins, en toutes circonstances, même quand c'est totalement inapproprié. Pour la petite histoire, et c'est l'enseignant qui parle, ce domaine n'est quasiment plus enseigné, sinon de façon superficielle et trop théorique, ni à l'université, ni dans les grandes écoles, et ce depuis plus de dix ans. La vague de l'IA, la frénésie des NTIC, lui a été fatale. L'industrie des compilateurs a disparu du paysage industriel français, alors qu'elle était florissante dans les années 70-80s, grâce à la présence des constructeurs BULL et CII, et aux besoins du Ministère de la Défense en langages temps-réel (LTR1, 2 et 3, puis Ada). Il n'y a plus de consultant en assez grand nombre dans les SSII, dotés d'une réelle expérience pratique sur le sujet, pour répondre à la demande, si elle se manifeste.

- Il faut donc s'attendre à d'énormes difficultés de compréhension de l'approche MDA, et plus encore de sa mise en œuvre dans les projets, avec comme conséquence inéluctable le rejet de la technologie, et une perte de confiance accrue dans l'industrie française du logiciel, au grand bénéfice de l'off-shore.

Comprendre la problématique de la machinerie informationnelle que nous n'avons fait qu'ébaucher requiert tout à la fois de bien saisir les nuances syntaxe versus sémantique, représentations concrètes versus représentations abstraites, langages internes versus langages externes, sans perdre de vue l'entité globale que constitue la machine et la finalité qui lui a été assignée. Inventer une machine revient à inventer un langage, dont les instructions sont des transactions ACID opérant sur un modèle de données explicite. Le challenge de l'architecte d'un projet d'interopérabilité est de définir un ensemble de machines informationnelles coopérantes qui, in fine, seront exécutées sur des plates-formes informatiques, mais cependant suffisamment abstraites pour pouvoir évoluer indépendamment des plates-formes, et inversement. La stabilité et l'orthogonalité conceptuelle du jeu d'instructions de la machine est le seul moyen pour répondre rapidement à un monde socio-économique en perpétuelle transformation.

- Ceci présuppose d'avoir su modéliser la notion de plate-forme, et en particulier le volet exceptions, faute de quoi, on ne sait pas ce que générer veut dire ! Dans ce domaine, tout reste à faire, mais là encore, la disparition programmée des architectes système, vu l'évolution du tissu industriel français, ne va pas faciliter le développement de l'approche MDA.

La technologie existe pour traiter correctement le problème, mais là encore cela nécessite d'avoir compris en profondeur les principes de construction de telles machines. Ce n'est évidemment pas avec des consultants n'ayant qu'une vague teinture informatique que les DSI pourront espérer passer l'obstacle.

Le dernier point, probablement le plus important, concerne l'IVVT de l'interopérabilité et l'évolutivité globale d'une fédération de systèmes inter-opérant. Nous n'avons fait que l'évoquer.

Fonctionnellement, on peut rêver à un monde massivement interconnecté. Cela présente certainement un intérêt économique, dans une économie mondialisée, du point de vue des fournisseurs de technologie, dans une logique pilotée par l'offre. Du point de vue des utilisateurs, il faut bien sûr que tout cela fonctionne correctement, dans tous les cas de figure, sans détruire l'équilibre coût, qualité, fonctionnalité, délai.

Comme on l'a vu, interopérer revient à créer de la complexité, dans le bon sens du terme. C'est également un moyen de contrôler la croissance des systèmes d'information, en évitant les duplications de fonctions qui créent toujours de l'incohérence.

Une règle d'ingénierie salutaire serait de ne jamais s'engager dans une solution d'interopérabilité si l'on n'a pas réfléchi préalablement à la question duale de l'intégration, et de reconsidérer l'ensemble de la problématique du point de vue de la testabilité de l'architecture globale envisagée.
Si l'on n'a pas une réponse claire à la question des tests et des défaillances, c'est à dire à l'ensemble du domaine qualité qui traite de la sûreté du fonctionnement des systèmes, mieux vaut ne pas partir à l'aventure, ou alors préparer le carnet de chèques. La problématique nouvelle introduite par l'interopérabilité est la sémantique, ce qui fait qu'une défaillance peut résulter de comportements d'acteurs dont la sémantique n'a pas été modélisée correctement, ou bien qu'entre le moment où cela a été fait, et la manifestation de la défaillance, l'environnement a changé. Il faut donc que la traçabilité intègre les acteurs et les comportements attendus, y compris et surtout, en cas de défaillance, ce qui devrait refreiner les ardeurs modélisatrices et expliciter les vertus d'une saine sobriété.
Là encore, toute la technologie existe pour résoudre ce type de problème. Mais avec l'interopérabilité, le temps de l'amateurisme est définitivement révolu. Les informaticiens doivent se comporter en ingénieurs responsables de leurs actes, quand bien même il s'agisse d'abstractions, mais encore faut-il qu'il s'agisse vraiment d'informaticiens correctement formés ! Les décideurs doivent accepter de payer le prix de la qualité.

Post-scriptum : L'ingénieur et le bricoleur

Texte de C.Lévi-Strauss, dans *La Pensée sauvage*, Plon.

…
Le bricoleur est apte à exécuter un grand nombre de tâches diversifiées ; mais, à la différence de l'ingénieur, il ne subordonne pas chacune d'elles à l'obtention de matières premières et d'outils, conçus et procurés à la mesure de son projet : son univers instrumental est clos, et la règle de son jeu est de toujours s'arranger avec les « moyens du bord », c'est-à-dire un ensemble à chaque instant fini d'outils et de matériaux, hétéroclites au surplus, parce que la composition de l'ensemble n'est pas en rapport avec le projet du moment, ni d'ailleurs avec aucun projet particulier, mais est le résultat contingent de toutes les occasions qui se sont présentées de renouveler ou d'enrichir le stock, ou de l'entretenir avec les résidus de constructions et de destructions antérieures. L'ensemble des moyens du bricoleur n'est donc pas définissable par un projet (ce qui supposerait d'ailleurs, comme chez l'ingénieur, l'existence d'autant d'ensembles instrumentaux que de genres de projets, au moins en théorie) ; il se définit seulement par son instrumentalité, autrement dit et pour employer le langage même du bricoleur, parce que les éléments sont recueillis ou conservés en vertu du principe que « ça peut toujours servir ».
…

ANNEXE 1 : NATURE DES COUPLAGES

Il est commode de distinguer trois niveaux de couplage, d'intensité décroissante : le couplage par les traitements, le couplage par les données, le couplage par l'ordonnancement.
Chacun de ces couplages peut prendre différents aspects, ce qui permet de graduer les niveaux de dépendance fonctionnelle des services.
Le meilleurs moyen d'identifier les couplages est de construire les matrices N2 des services résultant des différentes traçabilités[64].
Le couplage par l'ordonnancement est la relation minimum qui puisse exister entre les services. Les seules informations à considérer sont les évènements qui déterminent l'ordonnancement possible des services.

Couplage par les traitements
Ce type de couplage concerne la séquence des opérations.
La matrice N2 des services permet l'identification immédiate de ce type de couplage.
Modalité de l'appel du service
L'appel du service peut être synchrone (opérations séquentielles) ou asynchrone (opérations en parallèles via SEND et RECEIVE).
Dans le cas synchrone, le service appelé doit être disponible ; l'appelant s'arrête et attend le résultat, ce qui peut conduire à des situations de blocage si le service n'est pas disponible.
Modalité de répartition (aspect « remoting »)
Le service appelé peut être local ou distant
Dans le cas d'un appel local les données peuvent être partagées ; dans le cas d'un appel distant, le contexte de l'appel doit être transmis à l'appelant. Les données contextuelles peuvent être communiquées via les modalités propres aux données (ci-dessus).
Il faut distinguer les quatre cas {synchrone, asynchrone}×{local, distant} car les sémantiques et les caractéristiques non fonctionnelles sont totalement différentes. Il en ira de même pour les impacts.

Règle
Ce type de couplage implique un mécanisme unifié de gestion des codes retour et d'émission de messages d'erreurs (via un dictionnaire de messages).

Traçabilité des codes retour et des messages d'erreurs
On réalise une matrice {liste des services}×{liste des codes retour/messages d'erreur} en indiquant dans chaque case l'état du traitement {gravité de l'erreur, modalités de reprise, dialogue opérateur ou avec un automate d'administration, etc.}.

64 Cf. JP.Meinadier, *Ingénierie et intégration des systèmes*, Hermès.

Appel séquentiel

Le schéma de principe, avec un diagramme de séquence, est le suivant :

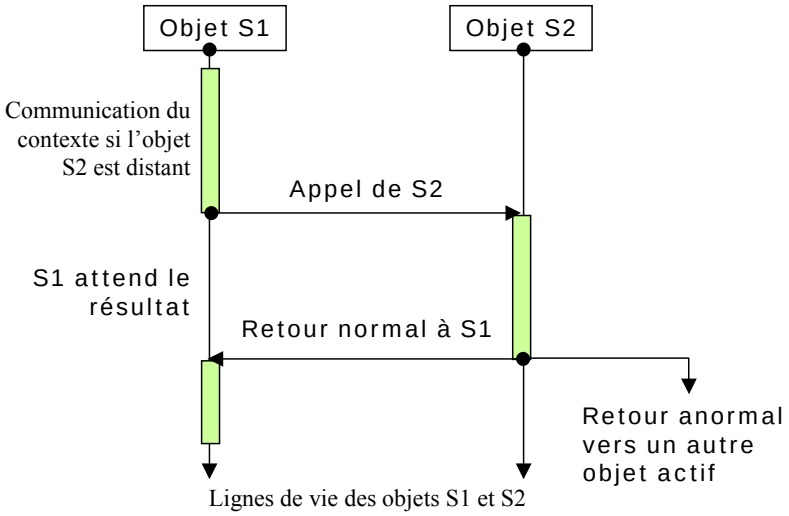


Figure 2.7 : Retours sur appel séquentiel

Le mode nominal de ce type d'appel est le retour à l'appelant. Si l'objet S2 est distant, la sémantique du retour est à préciser.

En cas d'anomalie de l'exécution de S2, le point de retour peut être différent de S1.

Appel asynchrone

Les messages qui s'échangent sont de type requêtes. Le retour peut être sur l'appelant ou sur un autre service.

Cas 1 : Notifications entre deux services. Le schéma de principe est le suivant :

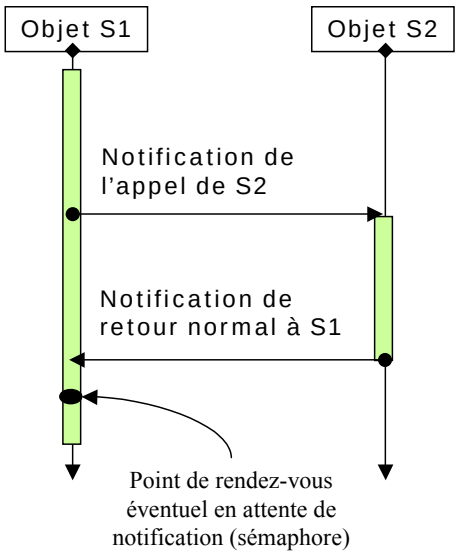


Figure 2.8 : Retour sur appel asynchrone

Cas 2 : Notifications entre trois services. L'objet S1 envoie sa requête à S2 qui l'exécute ; S3 récupère le résultat . Le schéma de principe est le suivant :

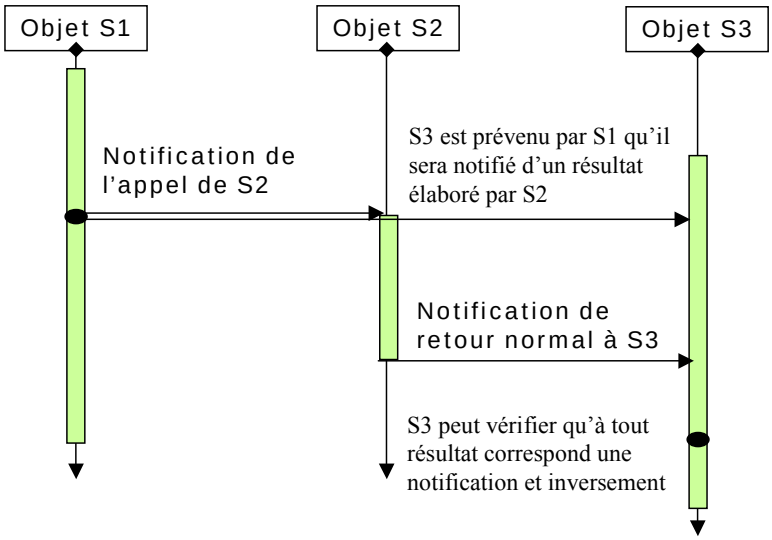


Figure 2.9 : Notification sur deux niveaux de service

Dans les deux cas, les anomalies rencontrées durant l'exécution de S2 doivent être prises en compte pour spécifier les modalités de retour (service fait, non fait, etc.).

Synthèse des mécanismes d'appels des services

	Appel local	**Appel distant**
Appel synchrone	C'est l'appel classique. Les modalités de passage des paramètres (référence, valeur, etc.) doivent être précisées.	Nécessite un RPC. C'est un protocole. Cf. SOAP, ou des protocoles du monde transactionnel.
Appel asynchrone	Est effectué via les mécanismes IPC disponibles dans le socle selon l'environnement choisi (cf. .NET ou J2EE)	Est effectué via les mécanismes d'interopérabilité du socle (de type messagerie, MOM, etc.)

Dans chacun des quatre cas, il faut préciser les modalités du retour : {FAIT, NON FAIT} ; dans le cas de NON FAIT, il faut préciser les causes en distinguant la ressource insuffisante (on peut attendre et réessayer plus tard), de la ressource indisponible (inutile de réessayer, c'est la panne !).

Couplage par les données

Deux services S1 et S2, ou deux composants, ou deux systèmes sont dans une relation de couplage par les données si, et uniquement si, la seule chose qu'ils ont en commun sont des formats de données. Les flots de contrôle dans lesquels ces services sont utilisés non aucun élément commun (i.e. les graphes d'appels des applications qui les utilisent sont non connexes) ni directement, ni indirectement.

Description des données

Il y a des données structurées (par exemple à la XML) et des données typées avec les mécanismes de typage des langages de programmation (ADT en objet, classes en UML, mappings IDL, etc.).

Utilisation des données

Les formats de données peuvent être utilisés de deux façons :

1. Utilisation pour référencer des données

Les services utilisent des descriptions qui doivent être identiques pour garantir leur intégrité. Ce cas correspond à des informations dupliquées dans les systèmes.

2. Utilisation pour accéder aux données

Ce cas correspond à des informations partagées. Les accès doivent être encapsulés, soit implicitement par les méthodes d'accès du SGF et/ou des SGBD utilisés, soit explicitement par des OBJET et les méthodes qui leurs sont associés (ce que nous avons appelé CRUD : Create, Retrieve, Update, Delete. Les règles de partages sont celles du SGF ou du SGBD ; en cas d'accès en mode transactionnel (respect des propriétés ACID) il faut un moniteur de transactions qui peut être soit un service informatique, soit un arbitre humain, soit un mélange des deux (cas du travail collaboratif).

Concernant les données accédées, il faut distinguer le cas ou le service à la vision directe des données (via le schéma de la base de données), du cas ou il a une vision indirecte et partielle des données (mécanismes de sous-schéma et de vues dérivées dans les SGBD, ou via une encapsulation ad hoc avec des méta-données).

Au total, il y a trois modes d'utilisation de la données, auxquels nous pouvons associer deux types de représentation : a) une représentation textuelle structurée à la XML (c'est une syntaxe concrète de la donnée) et b) une représentation typée au sens de la programmation telle qu'elle sera rencontrée dans les programmes. Soit six cas de couplage, a minima.

Les données textuelles structurées sont les moins contraignantes, au problème de performance près.

Dans le serveur de données (voir schémas ci-dessous dans les chapitres 3 et 4), les données sont accédées de façon indirecte via les méthodes d'accès (CRUD) associées au service. La restitution de la donnée peut être soit en XML (par défaut), soit typée dans le type de l'appelant, si le service dispose du convertisseur ad hoc ; dans ce cas l'impact en terme de dépendance est plus important, le couplage est plus fort.

NB : on pourrait imaginer une description de type texte libre (comme pour les moteurs de recherche) qui n'aurait d'intérêt que pour les parties purement textuelles des échanges.

Traçabilité

On réalise une matrice {liste des services}×{liste des données} en indiquant dans chaque case l'état de la données du point de vue de ses modes d'accès CRUD {input, output, input-output}.

Couplage par l'ordonnancement

La relation de couplage la plus faible concerne l'ordre ou la chronologie dans lequel les services doivent ou peuvent être exécutés. L'ordonnancement est réalisé via les évènements générés par les services et/ou par l'horloge du système.

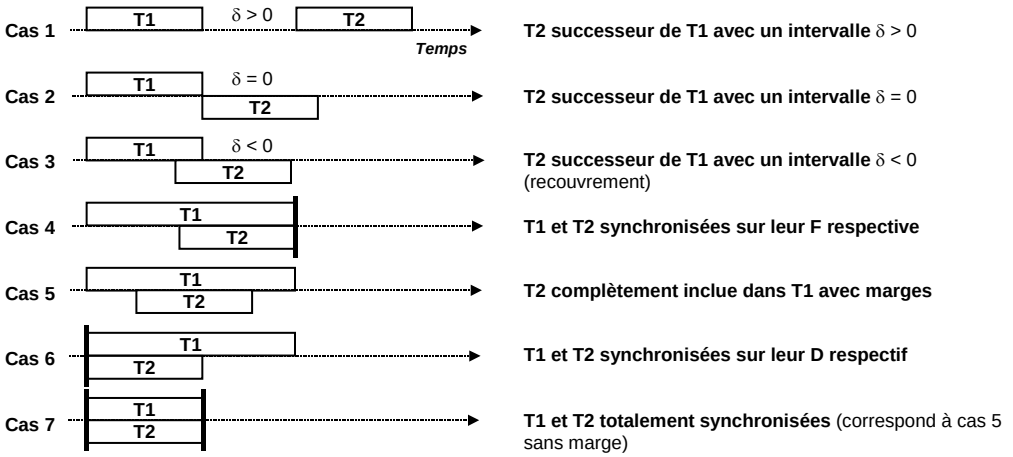
Exemple : S1 est réalisé (événement FIN NORMAL de S1) ; S2 peut démarrer. S1 est « planté » (événement FIN ANORMAL) ; le service de nettoyage doit être activé. S1 et S2 sont tous deux terminés (quel que soit l'ordre entre S1 et S2) ; S3 peut démarrer. Etc.

L'identification de ce type de couplage nécessite d'avoir une nomenclature des évènements générés par les services et les composant, ainsi que les drivers d'évènements correspondant. Ce type de description est possible dans des langages comme SDL (le standard UIT) ou BPML (Business Process Modeling Language) qui est un projet de norme OMG/W3C pour le workflow.

Types de dépendance temporelle

Le schéma ci-dessous donne les différents cas d'ordonnancement possibles entre deux tâches.

Légende : F = FIN de la tâche ; D = DEBUT de la tâche ; δ = intervalle temporel.



Les relations de dépendance par l'ordonnancement sont décrites soit à l'aide de diagrammes d'activités, soit à l'aide de diagrammes d'états, comme pour les workflow.

Traçabilité

On réalise une matrice {liste des services}×{liste des évènements} en indiquant dans chaque case l'état de l'évènement {producteur, consommateur}. Un service consommateur d'événements doit disposer de files d'attentes ad hoc qui sont des types de données (ADT) bien particulier qu'il faut identifier comme tel.

ANNEXE 2 : NOTION DE CHEMIN DE DONNEES

La notion de chemin de données est une des plus fondamentales qui soit en matière d'architecture de machine et de systèmes d'exploitation.

La transformation de l'information comme transduction

Dans la théorie de la mesure[65], en biologie, en théorie de l'information, on parle de transduction lorsque des signaux qui véhiculent une information sont retranscrits dans d'autres signaux acceptables par le destinataire ; par exemple une pression peut être convertie en courant électrique qui fera bouger une aiguille après amplification. Si il y a des parasites ou du bruit, ce que le destinataire interprétera sera erroné en tout ou partie. Le terme a été repris en informatique théorique, dans la théorie des machines à états finis[66], mais il est souvent assimilé à traduction. Cependant, une transduction est un terme plus fort qu'une simple traduction. Les commandes émises par le SIOC lorsqu'elles arrivent dans la têtes des acteurs de l'UA auxquels elles sont destinées doivent provoquer les effets escomptés, au moins au niveau de l'intention. Le manque de rigueur et de précision sémantique est une forme de bruit auquel les SIOC sont soumis, de par leur nature même. Ceci peut se produire à chacune des étapes ou l'homme intervient, soit directement, soit indirectement via les programmeurs qui ont compris plus ou moins bien l'information métier associée à la représentation informatique.

Dans les SIOC, la façon dont les données circulent entre les différents organes du système est donc très importante. Pour les besoins de simplicité de l'exposé, nous considérons trois cas : le puits, la source, et la transduction.

Puits d'information

Le système reçoit de l'information en provenance de sources externes : c'est une sémantique CRUD car il se met à jour (sous contrôle éventuel d'un opérateur).

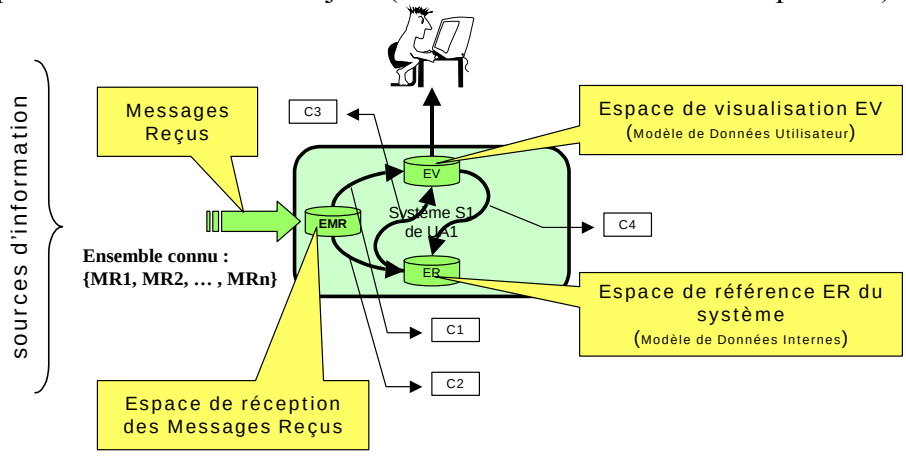


Figure 2.16 : Système en mode réception d'information

Seuls les chemins C1, C2, C3 et C4 sont actifs.

Source d'information

Le système envoie de l'information vers des sources externes : c'est une sémantique de type lecture CRUD (par rapport au destinataire).

---

65 Cf. G.Prieur et al., *La mesure et l'instrumentation – Etat de l'art et perspectives*, Masson, 1995.

66 Cf. P.Denning, & al., *Machines, languages, and computation*, Prentice Hall.

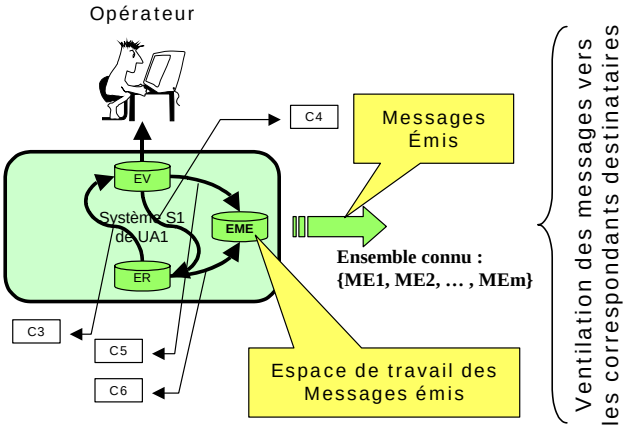


Figure 2.17 : Système en mode émission d'information

Seuls les chemins C3, C4, C5 et C6 sont actifs.

Transduction

Le système reçoit et envoie de l'information vers des sources externes : c'est une sémantique de mise à jour CRUD. Il y a traduction et re-formulation de ce qui rentre vers les UA subordonnées. C'est donc une vraie transduction, et non pas une simple traduction. La construction d'une COP (Voir chapitre 1 ci-dessus) est une transduction.
Ce mécanisme intègre les deux modes précédents avec une procédure d'enrichissement de l'information sous le contrôle des acteurs.
Le schéma ci-dessous est une exemple typique d'une chaîne de renseignement.

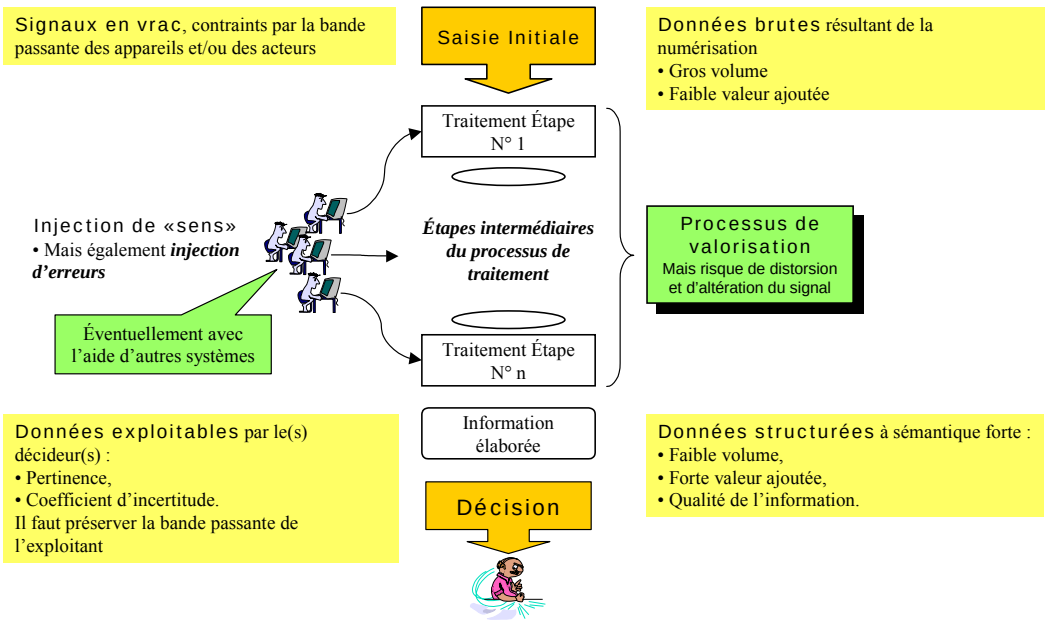


Figure 2.17a : Enrichissement sémantique d'une chaîne de traductions

Dans les deux cas qui suivent, on montre schématiquement comment le dispositif d'enrichissement fonctionne, avec plus ou moins d'aide d'opérateurs automatisés.
Tous les chemins sont actifs simultanément.

Cas 1

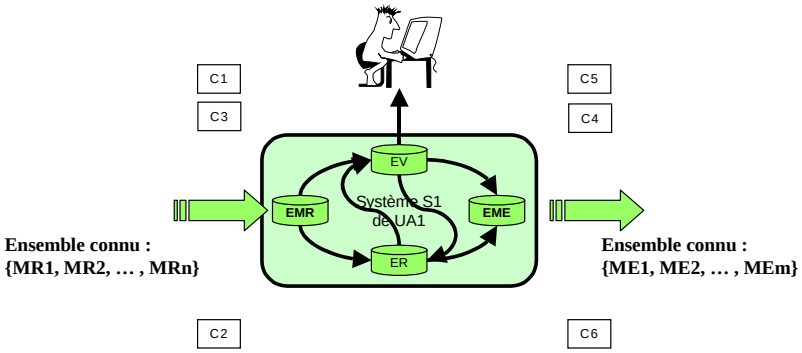


Figure 2.18 : Système en mode émission réception avec un contrôle opérateur

Cas 2

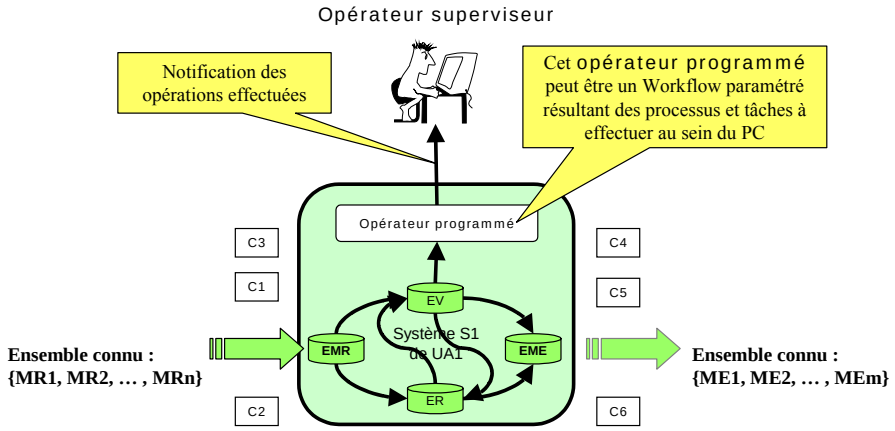


Figure 2.19 : Système en mode émission réception automatique avec un superviseur

L'opérateur programmé fait partie du mécanisme de surveillance qui, comme nous l'avons dit, participe pleinement à la sémantique des transformations effectuées par les transactions et services, avec toutes les difficultés de réalisation déjà signalées. Là encore, lucidité, sobriété, modestie doivent être au cœur des choix et des décisions de l'architecte, s'il veut respecter son client.

ANNEXE 3 : LES RELATIONS ENTRE ARCHITECTURE, COMPLEXITE, INTEGRATION ET PROGRAMMATION

Architecture

Le débat sur l'architecture logicielle est aussi ancien que celui concernant l'ingénierie du logiciel, dont il constitue un thème récurrent. Il suffit de relire les deux rapports du *NATO Science Committee*, *Software Engineering*, (en 1968 et 1969) pour s'en convaincre. On y trouve, entre autre, une brève communication de E.Dijkstra, *Complexity controlled by hierarchical ordering of function and variability*, très intéressante, et tout à fait révélatrice de problèmes qui sont toujours actuels. Des textes comme : *Cooperating sequential processes*, (1965) dans lequel le concept de sémaphore est rigoureusement défini, ou encore *THE multi programming system* dans lequel est définit le concept d'architecture en couches, gardent toute leur fraîcheur ; voir la recension : *The origin of concurrent programming*, de P.B.Hansen, Springer 2002, qui rassemblent de nombreux textes fondamentaux.
Le livre : *The mythical man-month* de F. Brooks, 1ère édition en 1975, toujours ré-édité depuis, peut à bon droit être considéré comme un ouvrage d'architecture de système logiciel (F. Brooks fut l'architecte de l'OS de la série 360 d'IBM). Tout architecte et/ou chef de projet soucieux de son art devrait l'avoir à portée de main ! Idem pour le livre d'H. Simon, *The sciences of the artificial*, 1ère édition 1981, toujours ré-édité, mais d'une toute autre nature que celui de Brooks. La 3ème édition (1996) contient un chapitre 8, *The architecture of complexity : hierarchic systems*, qui fait écho à ce qu'écrivait Dijkstra 30 ans plus tôt, d'une très grande profondeur car il ébauche une analyse quantitative entre les trois aspects qui nous préoccupent ici (données, transformations, événements).
Depuis les années 90s, le thème ***Software Architecture*** est à la mode tant au niveau des publications, nombreuses mais de qualité très inégale, que des manifestations.
Dans le livre de M.Shaw, D.Garlan, *Software Architecture*, Prentice Hall 1996, on trouve une définition (page 3) « *The architecture of a software system defines that system in terms of computational components and interactions among those components. … Interactions among components … can be simple … or complex and semantically rich, as client-server protocols, data-base-accessing protocols, asynchronous events multicast, and piped streams. … More generally, architectural models clarify structural and semantic differences among components and interactions.* Etc. … Etc. »
Définition intéressante, certes, mais combien réductrice ! Où est la liberté de choix de l'architecte, quelle est sa stratégie, que doit-il optimiser ? La définition ne le dit pas, ni d'ailleurs le livre. Notons cependant que le terme *computational components* renvoie à la notion de calcul (i.e. *business computing*), et donc de machine.
La littérature NCW et C4ISR-DODAF n'a évidemment pas échappé à cette vague. Pour la terminologie, elle s'appuie souvent sur le corpus de normes de l'IEEE *Software Engineering Standards Collection*, ce qui est bien, mais pas vraiment plus utile en pratique. On y trouve, des définitions comme : « les architectures fournissent des mécanismes permettant de comprendre et de manager la complexité » ou encore : « l'architecture décrit la structure des composants, leurs relations, les principes et les directives pilotant leur conception ».

Ce que toutes ces définitions révèlent, est qu'il est très difficile, voire impossible et/ou vain, de parler de l'architecture dans l'abstrait. L'architecture réfère toujours à quelque chose de concret. Brooks s'est avant tout préoccupé de l'architecture des systèmes d'exploitation, et il cultivait la métaphore des cathédrales : avec le temps, et les problèmes rencontrés par les premiers systèmes d'exploitation, l'expression est devenu plutôt péjorative. Son livre contient quelques recommandations très fortes comme :
« *By the architecture of a system, I mean the complete and detailed specification of the user interface* » ; aujourd'hui, on dirait les API, fléchant ainsi l'importance cruciale des interfaces.
« *Representation is the essence of programming. …The data or tables …is where the heart of a program lies* » ; aujourd'hui on dirait que le modèle de données est le cœur du système.
Aucun architecte de systèmes d'information sérieux, ne renierait ce genre de recommandations, et encore moins ceux en charges d'un C4ISR.
Le problème fondamental de l'architecte est d'organiser les communications, non seulement entre les composants constitutifs du système, mais également entre les équipes en charge de la réalisation de ces composants. L'architecture du système et de son logiciel rétro-agit sur l'architecture du processus de développement, et vice et versa ; stabiliser cette relation est l'une des tâches les plus complexes qui incombe à l'architecte.
Parmi les équipes, celles en charge de l'intégration des composants a un rôle bien particulier ; elle ne développe pas au sens classique du terme, mais elle assemble les composants dans un ordre qui n'est pas quelconque. Il est évident que l'architecture du système inclut une stratégie d'assemblage, faute de quoi il est certain que la construction risque d'avoir des problèmes de fiabilité dans ses étapes intermédiaires. La métaphore de la cathédrale reprend ici tout son sens : toutes les pierres (i.e. les composants logiciels) ayant été taillées correctement, (c'est très difficile pour les clés de voûtes, les rosaces, les ogives), le problème est de : comment les assembler sans que tout s'écroule en cours de montage (grâce aux échafaudages qui sont aussi des éléments d'architecture) ?
L'épitaphe de Robert de Luzarche, architecte de la cathédrale d'Amiens, le qualifie comme « docteur es-pierres » ! tout un programme.
On peut compléter les définitions précédentes de l'architecture, par une définition beaucoup plus opérationnelle intégrant la problématique projet et donnant un critère de terminaison :

Règle : l'architecture dans une perspective projet

- L'architecture d'un système est terminée quand, dans le projet de réalisation chacun sait ce qu'il doit faire (aspect descriptionnel et fonctionnel de l'architecture), comment il doit le faire (aspects non fonctionnels prenant en compte l'environnement du système, i.e. l'écosystème complet du projet) compte tenu des contraintes économiques de coût, qualité et délai.

Dans cette définition, « chacun » peut être un individu, une équipe nominale (≈ 7 personnes ± 2 est une bonne pratique), un ensemble d'équipes partageant une même finalité technique et/ou métier. La vision hiérarchique du système, conformément aux bonnes pratiques de l'ingénierie système, est fondamentale : c'est la structure la plus simple qui permet de travailler efficacement. Toute autre structure doit être évaluée en terme de risque, en particulier combinatoire.

Ce qui a été dit pour l'équipe d'intégration est parfaitement pris en compte dans cette définition. Cette équipe teste et assemble, avec comme critère la finalité d'emploi du système : pour travailler, elle a besoin des API, du plan d'assemblage et de scénarios d'emploi représentatifs de situations réelles, y compris en cas de défaillances qu'il faudra caractériser. Si elle n'en dispose pas, l'architecture n'est pas terminée, il y a donc un risque considérable à démarrer la réalisation.

Dans notre définition, on ne dit pas ce qu'est l'architecture du système dans l'abstrait, on se contente de dire, par rapport au projet de réalisation, quand le travail d'architecture pourra être déclaré fini. Dans cette approche, la réalité est le projet et son écosystème, et c'est dans le cadre du projet que se développe l'architecture, ce qui est une bonne façon d'éviter les dérapages conduisant à des développements inutiles : si l'on ne sait pas intégrer ou maintenir, ou comment former les usagers, mieux vaut s'interroger avant d'être face à ces échéances.

Sur cette base, on peut alors faire jouer des critères d'optimalité concernant la taille du système, son coût, sa qualité, le délai de la réalisation de la 1ère version, sa durée de vie (combien de versions est-il raisonnable de prévoir), le délai d'une intégration complète ou partielle, etc.

Compte tenue de la variété de ces différents critères, on sait qu'il n'y a pas de solution unique au problème posé. La situation est tout à fait analogue à celle du second théorème de Shannon qui montre qu'il existe toujours un moyen de compenser les erreurs résultant du bruit du canal de communication (dans notre cas, le canal est le processus de développement lui-même) par un codage ad hoc, mais que la construction de ce code (dans notre cas, le système qualité associé au processus de développement) est à réaliser au cas par cas.

Plus qu'une chose, ou un concept, réductible à un nom, l'architecture est un processus étroitement couplé à son environnement socio-économique. L'ordre de prise en compte des différentes problématiques, et les transformations opérées sur les différentes représentations qui vont alimenter le processus de programmation (« l'implémentation », dans le rapport du *NATO science committee*) est fondamentalement non linéaire et résulte des interactions entre les acteurs.

On sait que prendre en compte trop tardivement la problématique de l'erreur, ou ne pas se préoccuper assez tôt des performances conduit droit à l'échec ; mais il est très difficile de s'en préoccuper sans une première formulation du système, d'où les retours arrière et ipso facto, la non linéarité. Ceci vaut pour toutes les caractéristiques non fonctionnelles.

D'un point de vue théorique, c'est un problème de stabilité ou de point fixe :

$$\text{Arch}(\text{Arch}(x)) \rightarrow \text{Arch}(x)$$

Les étapes successives ne doivent pas altérer le schéma initial, mais le compléter; faute de quoi, le processus sera divergent.

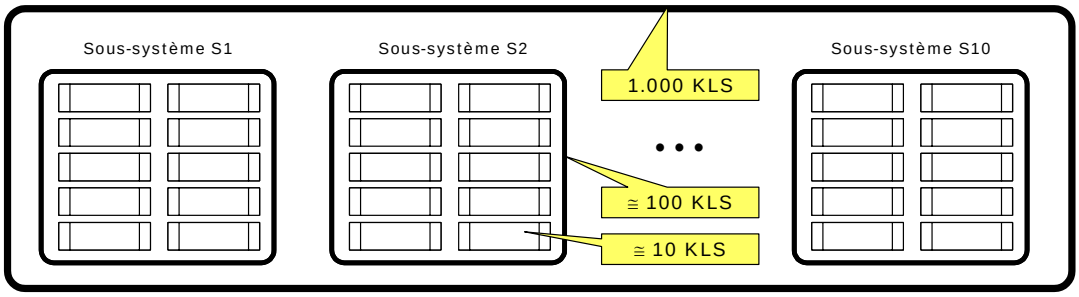
L'architecture est un processus d'optimisation multicritère, où l'on recherche des *Mini-Max* sur les critères CQFD, dont le résultat est une description textuelle et/ou graphique plus ou moins formalisée. C'est un jeu, au sens de la théorie des jeux, où l'ordre des évènements influe sur la solution adoptée in fine, et d'une certaine façon détermine la trajectoire du projet.

Si les données constituent le cœur du système, comme le proclame Brooks à juste titre, et induisent les traitements qui en constituent le cœur fonctionnel, ce sont les événements, résultant de la non linéarité du processus, qui vont en caractériser la représentation programmatique qui à l'instant t constituera la livraison. La qualité des événements et l'ordre dans lequel ils vont être pris en compte par l'équipe de conception, sont la matière première des décision architecturales qui, au final,

constituera non pas LA solution, mais UNE solution, qui elle même produira son lot d'événements conduisant à une deuxième version, et ainsi de suite. L'architecture initiale et son histoire événementielle détermine la capacité évolutive et adaptative du système, et d'une certaine façon sa durée de vie.

Architecture et complexité

Pour illustrer quantitativement la problématique considérons un système formé de 1000 composants logiciels élémentaires.
Pour la simplicité des calculs on considère un système de 1.000.000 de lignes de code source (LS), soit, en millier de LS, 1.000 KLS (NB : cette taille correspond à un « gros » système d'information comme il en existe aujourd'hui des dizaines dans les entreprises).
Ce système, conformément à la terminologie de l'ingénierie système est formé de : 10 éléments comportant chacun 100 KLS (ce sont des ***sous-systèmes*** et/ou des ***applications*** constitutifs du système complet) ; chaque sous-système est formé de 10 modules/packages comportant chacun 10 KLS ; soit au total 100 modules/packages à intégrer pour constituer le système complet.
Le schéma ci-dessous donne la topographie d'un tel système.



Le nombre d'« intégrats[67] » dans une telle structure hiérarchique est donc égal à :
100 + 10 + 1 = 111 chaque intégrat regroupant 10 entités de rang inférieur (appelées sous-système/application, ou module, ou «composant » selon le niveau).
On fera l'hypothèse que les composants (on peut les assimiler à des objets) sont eux mêmes formés de « pièces/méthodes » logiciel de 1 KLS en moyenne (NB : c'est l'unité de comptage du modèle COCOMO) ; dans ce modèle l'équation d'effort pour un système de ce type est donnée par la formule : $Effort_{en\ hm} = 2{,}4 \times (Taille_{en\ kls})^{1{,}05}$).

- Le système complet comporte donc 1.000 « composants » élémentaires (ou 10.000 « pièces » selon le niveau de granularité considéré.
- La vision hiérarchique est donc : Système → Sous-système/application → Modules/packages → Composants/Objets → Pièces/Méthodes

Les composants logiciels sont le résultat du travail de petites équipes de programmation dont l'effort de réalisation moyen est de 2,5 ha par composant, ce qui correspond à une productivité industrielle moyenne de l'ordre de 350 LS par programmeur et par mois (lignes de code documentés et testées, conformément aux normes en vigueur). Une

[67] Correspond à la notion de « building blocks » ; du point de vue de l'intégration, ce sont des boites noires qui ne sont connues de l'intégration que par leurs interfaces.

pièce est le travail d'***un*** programmeur, mais un programmeur peut réaliser plusieurs pièces.
On remarquera que si l'intégration ne coûtait rien, le coût d'un tel système serait simplement de 100×2,5=250 ha. On peut vérifier la cohérence des coûts en comparant l'effort de développement de 1.000 KLS à celui de 100 fois 10 KLS, en utilisant la formule COCOMO ci-dessus !

NB : Pour tous les calculs d'effort avec le modèle COCOMO, on raisonne en année pleine, soit 1 année ≈220 jours ouvrés ; 1 jour ouvré = 8h brutes, avec 1hm de ≈20j (les congés, les 35 heures, … modifient substantiellement ces calculs !!! ; on ne travaille pas plus vite pour autant) ; pour plus de détail, voir nos deux ouvrages *Productivité des programmeurs*, et *Coûts et durée des projets informatiques*.

Le responsable de l'intégration du système S a le choix entre deux stratégies,

1. Stratégie N°1 (parfois appelée big-bang) : il intègre les 1.000 pièces d'un seul coup car il ne connaît pas la nature des relations qui peuvent exister entre les différentes pièces qui lui sont livrées ; dans cette stratégie, le responsable d'intégration reçoit donc 1.000 pièces, en vrac.
2. Stratégie N°2 : le système est structuré en couches/serveurs indépendant[e]s, structurées en blocs de 10 entités selon le niveau, jusqu'à arriver au système complet ; dans cette stratégie, le responsable d'intégration reçoit toujours 1.000 pièces, mais il sait les assembler par paquets de 10, jusqu'à constitution de l'intégrat final. Il organise le processus d'intégration en fonction des informations qui lui ont été communiquées par l'architecte du système.

On remarquera qu'en terme de combinatoire, la stratégie S1 est de l'ordre 1.000 ! (i.e. factorielle 1.000, qui est un nombre immense), alors que S2 est de l'ordre $10 \times 100! + 10!$, également très grand, mais beaucoup plus petit en relatif.
Le niveau de test des pièces élémentaires (obtenu par la mise en œuvre d'une politique rigoureuse de tests unitaires) est tel que l'on peut estimer leur fiabilité ***f*** à 0,99%, ce qui signifie que sur 100 exécutions de la pièce on observera, en moyenne, une seule défaillance ou anomalie (NB : les données d'entrée d'une pièce peuvent changer d'une exécution à la suivante, ainsi que l'environnement de la pièce, conformément au phénomène de dégénérescence qui se manifeste lors de l'exécution des pièces[68]).

Fiabilité des chaînes de liaisons

La formule classique donnant la fiabilité d'une chaîne de liaison de longueur *n* qui regroupent des agrégats (voir figure 2.6), est :

$$F(n) = f^n$$

soit pour n=10 → F=0,90 ; n=100 → F=0,37 ; n=200 → F=0,13 ; n=500 → F=0,007 et n=1.000 → F=0,00004
En valeur brute, la fiabilité tend donc inéluctablement vers 0 !
Il faut toutefois faire très attention à l'interprétation de ces valeurs dans la réalité des modes d'exécution et d'emploi des systèmes informatiques ; c'est ce que nous allons maintenant esquisser.

La question de fond concerne l'interprétation du ratio des fiabilités relatives d'un agrégat de 10 éléments agrégés progressivement par paquet de dix (stratégie d'intégration N°2) sur un agrégat de 1.000 pièces (stratégie d'intégration N°1) ? Quel

[68] Cf. J.Printz, *Puissance et limites des systèmes informatisés*, déjà cité.

est le ratio d'effort de non régression dans l'une ou l'autre de ces stratégies, compte tenu du nombre moyen d'éléments sur lequel porte la non régression.
NB : On peut remarquer que dans la stratégie N°1, l'intégrat système est de 1.000 pièces ; et que dans la stratégie N°2, il y a 111 intégrats dont chacun comporte 10 pièces (élémentaires, ou déjà agrégées) ; dans ce dernier cas, la fiabilité du système complet est $F = \frac{f^{10}}{111}$, car la fiabilité — qui est équivalente à une probabilité — est additive dans ce cas, alors qu'elle est multiplicative dans l'autre.

Dans ce qui suit, le système S, au moment ou démarre l'intégration, contient un certain nombre d'erreurs résiduelles dont la distribution ne dépend pas de la stratégie d'intégration. Ce taux dépend de l'efficacité des tests unitaires faits sur les différentes pièces qui ont conduit à une fiabilité de 0,99 pour chaque pièce.
Il est intuitivement évident que la stratégie S1 est plus coûteuse que la stratégie S2, mais la vraie question est de savoir de combien ? qq. % ? ou 1 ou 2 ordres de grandeur, soit ×10, ou ×100 plus.
Ce qui suit ébauche la ligne de raisonnement, sur la base des hypothèses simplificatrices que nous nous sommes données, et détermine les ordres de grandeur.
NB : pour des raisonnements[69] plus fouillés, voir en particulier le chapitre 7.2 du livre *Puissance et limites des systèmes informatisés* et *Productivité des programmeurs*.

Le système le moins fiable occasionne en moyenne plus de défaillances ; on peut donc s'attendre à ce que le coût des interventions avec S1 soit plus élevé que avec S2.
Dans la stratégie S2, l'intégration se fait en trois étapes :

- ✓ 1ère étape : 10 « pièces » de fiabilité 0,99 sont intégrées pour constituer un module auquel est associé un jeu de test qui permettent d'obtenir une fiabilité de 0,99. Ce travail est à faire 100 fois.
- ✓ 2ème étape : on regroupe les modules par paquet de 10 pour former les sous-systèmes et les applications auxquels sont associés de nouveaux jeux de tests qui permettent d'obtenir une fiabilité de 0 ,99. Ce travail est à faire 10 fois.
- ✓ 3ème étape : intégration finale des 10 sous-systèmes pour former le système complet, avec un nouveau jeu de tests qui permet d'obtenir une fiabilité finale de 0,99.

Le rôle des tests d'intégration est donc de ramener la fiabilité des intégrats à celle de leurs constituants élémentaires, et ce jusqu'à l'obtention de la fiabilité globale exigée par la qualité de service auquel le système doit satisfaire.
Si les 111 intégrats issus de la stratégie S2 ont tous la même fiabilité (soit 0,99), on est en droit de dire que la fiabilité de S issue de la stratégie S2 est égale à $\frac{f^{10}}{111} \approx 8 \times 10^{-3}$ (les intégrations des différents intégrats sont indépendantes, donc les probabilités sont additives). Le quotient des fiabilités s'interprète comme le nombre moyen d'interventions à effectuer sur S compte tenu de la stratégie d'intégration S2 rapporté à S1, soit $N = \frac{\frac{f^{10}}{111}}{f^{1000}} \approx \frac{8 \times 10^{-3}}{4 \times 10^{-5}} \approx 200$.

[69] Voir également un raisonnement très intéressant dans H.A.Simon, *The sciences of the artificial*, MIT Press 1996, 3rd edition ; en particulier le chapitre 8, *The architecture of complexity : hierarchic systems*.

Sous ces hypothèses, il y a 200 fois plus d'interventions avec la stratégie S1 que avec la stratégie S2, donc il y aura beaucoup plus de non régressions à effectuer avec S1 que avec S2.
Reste maintenant à estimer le coût moyen d'une intervention, dans l'une ou l'autre des stratégies d'intégration.
Avec S1, on a moins de scénarios, mais les scénarios sont plus longs à écrire et ils sont plus complexes à vérifier que avec S2 car les ensembles à considérer ont des cardinalités très différentes : 1000 versus 10 .
La taille de l'intégrat (i.e. le nombre de constituants élémentaires de l'intégrat) détermine la taille du/des scénarios de tests correspondant à cet intégrat. Lors d'un test, certaines pièces sont utilisées plus fréquemment que d'autres. Si l'intégrat contient ***n*** pièces, il faudra exécuter beaucoup plus que ***n*** pièces pour pouvoir les exécuter toute, au moins une fois ; ce qui complique à la fois l'écriture du scénario et sa vérification (donc cela augmente son coût). En effet, en l'absence d'information sur la structure de l'intégrat, il faut s'en remettre au hasard, et au phénomène de doublon que le hasard implique.
Avec la stratégie S1, on teste à l'aveugle en faisant complètement confiance au hasard ; on ne sait même pas quelles sont les pièces effectivement utilisées, et combien de fois elles l'ont été : on ignore tout de la distribution, et cette ignorance a évidemment un coût en terme de non qualité (ou d'effort additionnel, selon les points de vue).
Dans la stratégie N°2, le processus d'intégration est entièrement guidé par l'architecture (cf. le concept d'architecture testable). Si une défaillance apparaît on sait exactement dans quel intégrat parmi les 111 cette défaillance apparaît ; en conséquence, on en cherche la cause dans le sous-ensemble correspondant qui est beaucoup plus petit (10 au lieu de 1000).
Le processus peut être schématisé comme suit : on intègre par paquet de 10 de façon à ce que la fiabilité de l'intégrat ainsi obtenu soit égale à 0.99, soit :

$$Intégration\{P1,P2,\ldots,P10\} \xrightarrow[Coût]{} Intégrat(I1)$$

Le coût d'intégration correspond à l'écriture et à la vérification des scénarios pour obtenir une fiabilité de 0,99 ; c'est un critère d'arrêt d'intégration.
En prenant les valeurs extrêmes de statistique généralement admises[70] (2 à 5 heures versus 50 heures), il n'est pas absurde d'estimer le différentiel de coût de correction des défaillances dans S1 par rapport à S2 de l'ordre d'un facteur 10.

Conclusion: il y a beaucoup plus d'interventions de l'équipe d'intégration avec la stratégie S1 que avec la stratégie S2 (≈200), et ces interventions sont statistiquement beaucoup plus coûteuses (≈10).
La stratégie d'intégration est un résultat du processus de conception. L'absence de ce livrable fondamental est un risque très élevé de non respect des caractéristiques CQFD et une quasi certitude d'un manque de maturité de l'équipe projet, qui aura d'autres conséquenses.
Même si nos hypothèses sont simplificatrices pour la simplicité des calculs, on voit qu'une architecture bâclée ou naïve (en général, personne n'a le désir de nuire, mais avec la complexité, le bon sens est toujours pris en défaut) aura des conséquences catastrophiques au moment de l'intégration, ce qui se traduira par deux constats, aux

[70] Voir en particulier R.Grady, *Practical software metrics for project management and process improvement*, Hewlett-Packard professional book, Prentice Hall.

symptômes cependant bien visibles si l'on observe correctement le processus de développement au moyen d'un SAQ étudié pour :

1. Allongement considérable de la durée de l'intégration (et en conséquence de son coût), car il est très difficile d'organiser le parallélisme des tâches d'intégration.
2. Défaut de qualité flagrant (performance, fiabilité, sûreté de fonctionnement, … entres autres) dès les premières installations, auquel il faudra rajouter, plus tard, de grandes difficultés d'évolution et maintenance.

Une économie sur l'effort de conception se paye toujours très chère en intégration et/ou en qualité.
On voit également l'importance de ce qui est fait en matière d'automatisation de la non régression, car la réduction du coût correspondant est d'un ordre de grandeur, au moins.

Architecture et programmation

Comme indiqué précédemment, l'un des rôles de l'architecte est de préparer le travail des programmeurs. Le référentiel d'architecture, quel que soit sa forme et son degré de formalisation, doit leur permettre de déterminer la nature des fonctions qui sont à programmer ainsi que la nature des contraintes à prendre en compte pour savoir comment les programmer, conformément aux exigences comportementales auxquelles doit satisfaire le système complet.
En suivant la terminologie de la norme ISO/CEI 9126, *Caractéristiques qualité des produits logiciels*, chaque sous-système/application, chaque module/paquetage, chaque composant/objet et chaque pièce/méthode sera déclinée en : Functionnality, Usability, Reliability, Performance (i.e. efficiency), Maintenability (i.e. serviceability), Portability (i.e. evolutivity) qu'il est commode de résumer par le sigle FURPSE ; la partie URPSE constitue les caractéristiques non-fonctionnelles. Ceci veut dire que dans le code source correspondant, nous allons trouver le code fonctionnel nominal qui assure la fonction auquel sera associé du code, en plus ou moins gros volume, pour assurer chacune des caractéristiques non-fonctionnelles. L'ensemble est généralement très intriqué, ce qui fait qu'il est très malaisé de dire si telle instruction fait partie de telle ou telle caractéristique. Une même instruction peut satisfaire simultanément plusieurs caractéristiques.
Selon le niveau des exigences, à compétence de programmeur égale, le code est plus ou moins coûteux à réaliser. En suivant l'approche du modèle d'estimation COCOMO, on considère trois catégories de programmes auxquelles vont correspondre trois équations d'effort (ce sont celles du modèle basique) :

- Equation 1 : $Effort_{en\,hm} = 2{,}4 \times \left(Taille_{en\,kls}\right)^{1{,}05}$, soit pour 100 KLS : 302 hm
- Equation 2 : $Effort_{en\,hm} = 3{,}0 \times \left(Taille_{en\,kls}\right)^{1{,}12}$, soit 521 hm
- Equation 3 : $Effort_{en\,hm} = 3{,}6 \times \left(Taille_{en\,kls}\right)^{1{,}20}$, soit 904 hm.

Toute chose égale par ailleurs, il y a un facteur trois selon la nature du code à réaliser.
Résumons la ligne de raisonnement qui justifie le bien fondé de cette catégorisation (pour une présentation en détail, nous renvoyons à nos ouvrages déjà cités) :

Règle : l'architecture dans une perspective programmation

- Tout programme comprend une ***composante réactive*** (i.e. sa structure de contrôle qui agit sur le flot ; régie par l'équation 3) et une ***composante transformationnelle*** (i.e. ce qui régit les changements d'état de la mémoire) qui peut être ***simple*** ou ***complexe***, régie par les équations 1 et 2.

La composante transformationnelle peut elle-même être décomposée en deux sous-composantes :

1. ***Transformation simple*** (une transcription) régie par l'équation 1, comme par exemple une règle de gestion métier dont la programmation n'est qu'une traduction en langage informatique d'une règle exprimable en langage naturel propre au métier ; c'est l'idée fondamental du langage COBOL, à l'origine. Une telle programmation peut être validée par une simple comparaison aux textes définissant les règles métiers.
2. ***Transformation complexe***, régie par l'équation 2, à l'aide d'algorithmes, d'un ensemble d'états pouvant présenter une grande variabilité, et un volume quelconque, en un résultat fini (traitements statistiques, optimisation, traduction, paramétrage, etc.). Pour être intéressant, au delà du simple cas particulier, l'algorithme doit présenter un caractère de généralité plus ou moins poussé, ce qui permet d'englober un grand nombre de cas en une seule formulation abstraite. L'abstraction est l'essence du travail de programmation. Valider un algorithme, c'est démontrer que tous les cas, et rien que les cas (i.e. la robustesse, ou résilience, de l'algorithme), auxquels il s'applique sont effectivement pris en compte, quelle que soit la méthode de démonstration adoptée, expérimentale à l'aide de tests, ou formelle avec un démonstrateur.

Pour l'architecte, la question est donc de savoir, et de pouvoir, construire des blocs de code de caractéristiques homogènes minimisant la composante réactive, et surtout de donner des règles et des guide de programmation permettant aux programmeurs d'agir conformément à la logique de construction du système.

Quiconque a pratiqué la programmation « en grand » sait que cela est un exercice non trivial qui nécessite une compréhension profonde de la sémantique du programme par rapport au contexte système dans lequel le programme opère.

La composante réactive est, in fine, formée des instructions de contrôle du flot :

> `IF … THEN … ELSE …` , `GOTO`, `CALL` *`fonction`*, `EXIT`, les boucles, la communication inter-processus, les pilotes d'événements, etc. Mais l'inverse n'est pas nécessairement vrai.

L'instruction IF peut servir à construire des assignations conditionnelles que l'on aimerait pouvoir écrire comme :

> *a* `=` `IF` *`cond`* `THEN` *b* `ELSE` *c*

pour dire que *a* vaut *b* ou *c* selon la condition.

La même remarque vaut pour une table de décision.

Par ailleurs, un automate de contrôle se représente très bien par un tableau correspondant au graphe de l'automate, donc sous forme de données.

La simple considération des instructions est insuffisante, et de trop bas niveau, pour travailler sur la structure de contrôle, en particulier lors des tests. De plus, la structure du flot de contrôle peut elle-même être hiérarchisée, mais la hiérarchie correspondante n'est pas une donnée évidente. Dans un langage, par exemple, on distingue depuis toujours le niveau lexical du niveau syntaxique, mais dans la réalité la frontière n'est pas toujours aussi nette : il y a des abréviations, des onomatopées, des locutions formant bloc, des métaphores, etc. Cela sera encore plus vrai dans le cas des workflows associés à un métier particulier.

Règle : architecture de la composante réactive

La composante réactive est elle même une structure que l'on peut organiser en hiérarchie. L'identification des hiérarchies nécessite une compréhension profonde de la mécanique des enchaînements en distinguant soigneusement ce qui est purement fonctionnel métier (aspect PIM) et ce qui est induit par la logique informatique (aspect PSM). La hiérarchie du contrôle permet de visualiser l'histoire de la transformation, de faciliter le diagnostic en cas de défaillance (la trace est le langage de l'automate correspondant) et donc de permettre l'optimisation de l'effort de VVT.

Si l'on prend comme exemple de programme un traducteur, comme l'approche MDA en présuppose de nombreux et de toute nature, à toutes les étapes de la transformation des modèles, on sait, pour de tels programmes, séparer la composante réactive de la composante transformationnelle (si tant est que l'architecte ait été formé correctement en techniques de compilation). La taille de la composante réactive est de l'ordre de 10 à 15%, voire moins en utilisant des automates de reconnaissance. La composante transformationnelle peut être très algorithmique (cas des compilateurs) où relativement simple (cas d'un générateur d'applications, où il s'agit généralement de transcription, sans optimisation, ni mémorisation de ce qui a déjà été traduit, i.e. pas de gestion de cache).

NB : Il est remarquable de noter que le pattern MVC dont nous avons fait le modèle de la machine informationnelle, est parfaitement conforme au pattern général des traducteurs.

La structure des coûts d'un tel programme est :

Coût total = Coût de la composante réactive + Coût de la transformation

Soit, pour 100 KLS : 57hm + 270hm = 327hm.
Si l'architecte (et les programmeurs) ne connaissent pas les techniques de compilation, ils peuvent cependant réaliser un traducteur dans lequel la composante réactive sera étroitement intriquée avec la composante transformationnelle, ce qui fait que quasiment tout le code doit être assimilé à du réactif ; la proportion s'inverse :
Soit, pour 100 KLS : 797hm + 27hm = 824hm.
L'amplification du coût est de 2,5 ; et la durée sera en conséquence !
De plus, le programme sera très difficile à maintenir et à faire évoluer. Il sera également le reflet des idiosyncrasies de son concepteur, et de ce fait très difficile à transmettre à des tiers ayant une tournure d'esprit différente. Le système contient une bombe à retardement du point de vue du coût d'acquisition total (TCO).
Dans le cas de l'architecte expert en technique de compilation, on peut raisonnablement supposer que s'il manie parfaitement le concept d'automate, il a su également décomposer la partie transformationnelle en actions indépendantes (c'est une pure approche machine abstraite), ce qui fait que les 90 KLS de transformation peuvent être assimilée à 90 actions indépendantes de 1 KLS chaque (pour la simplicité du calcul), ce qui ramène le coût de la transformation à 90×2,4 = 216, soit un gain de 54hm. Enfin, *last but not least*, on peut également penser qu'une telle organisation du code facilite l'identification d'actions communes factorisables, et donc l'émergence de nouvelles abstractions ; c'est ce que B.Boehm, dans le modèle COCOMO II, appelle le facteur d'échelle. Les 90 KLS initialement estimées ne sont peut-être que 70, d'où le coût final de la transformation : 168hm.

L'effet d'une bonne architecture de programmation est donc pour ce cas d'école, au final, un coût instantané de 225hm, à comparer au 824hm (soit un facteur 3,7), sans parler du coût récurrent de maintenance et d'évolution.
Le même raisonnement vaut pour la séparation du code transformationnel en transcriptions simples et cas particuliers, et en algorithmes. Un algorithme doit toujours être encapsulé de façon à pouvoir être remplacer, le moment venu, par un nouvel algorithme satisfaisant mieux les exigences non-fonctionnelles ; seule l'interface d'appel doit rester invariante.

Conclusion : le travail d'organisation du système en sous-systèmes et modules autonomes (i.e. leur sphère de contrôle est explicite) pour contrôler le coût d'intégration système et la fiabilité de l'assemblage ainsi obtenu, doit être accompagné d'une réflexion approfondie sur la nature du code associé aux modules et composants correspondants.
Les différentes composantes réactives doivent être explicitées en fonction de la nature des interactions qu'elles régissent : services ou ressources de la plate-forme. Une ressource de mémoire persistante ou de mémoire non-persistante n'a pas les mêmes caractéristiques non-fonctionnelles : il faut donc les distinguer. Un service transactionnel centralisé (mettant en œuvre des transactions courtes, au sens base de données) ou distribué (mettant en œuvre des transactions longues, dans une logique métier) ne peut pas gérer les défaillances selon une logique uniforme ; les procédures de sauvegardes et de reprises seront nécessairement différentes. Un service métier défaillant peut être réparé en interrogeant l'opérateur humain ; ceci est impossible pour un service gérant des ressources matérielles. Les paramétrages seront également très différents, selon que l'on est du coté PIM, ou du coté PSM.
L'approche par machines abstraites permet non seulement d'organiser le systèmes en hiérarchie de machines coopérantes, mais également d'organiser le code en blocs de complexité homogène, évitant ainsi de transformer des pans entiers de code transformationnel en code réactif. Elle permet également de gérer au mieux la ressource programmeur en confiant la réalisation des composantes réactives et des composantes transformationnelles complexes aux plus expérimentés d'entre eux.