<table>
<tr><td>Type : MAN</td><td rowspan="3">Projet RNTL<br>MORSE</td><td>Réf : MAN-METHODOLOGIE</td></tr>
<tr><td>Date : 25/06/04</td><td>Sous projet n° : 1.1</td></tr>
<tr><td>Version : 1.2</td><td>Tâche n° : 1</td></tr>
<tr><td colspan="3">Auteurs : Maurice Assouline, François Bréant, Jean-Michel Couvreur, Frédéric Gilliers, Fabrice Kordon, Marc Richard-Foy, Grégoire Sutre, Jean-Pierre Vélu</td></tr>
<tr><td colspan="3">Editeur : Maurice Assouline</td></tr>
</table>

# Guide Méthodologique MORSE

## Liste des révisions

| | Paragraphe | Commentaire |
|---|---|---|
| *Version* : 1.0 (plan)<br>*Date* : 10/02/04<br>*Auteur* : Marc Richard-Foy | tous | Version initiale (plan) |
| *Version* : 1.1<br>*Date* : 16/04/04<br>*Auteurs* : MRF, FG, JPV, MAS, GS | 1, 2.1, 3.1, 3.4.8 | Insertion d'apports initiaux.<br>Styles MS-Word |
| *Version* : 1.2<br>*Date* : 25/06/04<br>*Auteurs* : FB, MAS | 3.4.8, tous | Insertion du § 3.4.8-"Vérification", mise en forme, corrections mineures. |

# Table des matières

**1 CONTEXTE** ..... **4**

1.1 LES GRANDES LIGNES DE LA METHODOLOGIE SAGEM (CYCLE EN Y+SPIRALE) ..... 4
1.2 MORSE ET MDA (MODEL DRIVEN ARCHITECTURE) ..... 5
1.3 MORSE ET DEMARCHE PAR PROTOTYPAGE ..... 6

**2 DOMAINE D'APPLICATION** ..... **8**

2.1 OBJECTIFS ..... 8
2.2 CARACTERISTIQUES ..... 10
2.3 CONTRAINTES ..... 10

**3 PROCESSUS** ..... **10**

3.1 VUE D'ENSEMBLE DES ETAPES DU PROCESSUS ..... 10
*3.1.1 Sous-processus 1 : obtention d'un modèle UML correct ..... 10*
*3.1.2 Sous-processus 2 : génération du programme ..... 13*
3.2 ECRITURE DU MODELE UML ..... 15
3.3 TRAÇABILITE ..... 15
3.4 FORMALISATION ..... 15
*3.4.1 Comment faire un modèle UML ..... 15*
*3.4.2 Comment appliquer le profil ..... 15*
*3.4.3 Comment séparer contrôle et métier ..... 15*
*3.4.4 Comment faire un cycle de formalisation ? ..... 15*
*3.4.5 Comment générer le LfP ..... 15*
*3.4.6 Prise en compte des erreurs ..... 15*
*3.4.7 Comment établir les propriétés à vérifier ..... 15*
*3.4.8 Vérification ..... 15*
*3.4.9 Génération de code (composants métier) ..... 18*
3.5 DIRECTIVES DE GENERATION DE CODE, DU MODELE… ..... 18
3.6 CONFRONTATION PRATIQUES EN COURS (SE POSITIONNER PAR RAPPORT…) ..... 18

**4 TRAITEMENT DE GROSSES APPLICATIONS (PASSAGE A L'ECHELLE)** ..... **18**

**5 GLOSSAIRE** ..... **18**

**6 ABREVIATIONS ET ACRONYMES** ..... **18**

**7 REFERENCES** ..... **18**

# 1 Contexte

## 1.1 Les grandes lignes de la méthodologie SAGEM (Cycle en Y+Spirale)

L'idée principale qui est à la base est celle d'un processus permettant le contrôle continu du développement. La démarche est incrémentale et consiste à réaliser plusieurs *prototypes* convergents vers le produit final. Un prototype est un exécutable non jetable pouvant être testé à l'aide de scénarios exhaustifs représentatifs du métier. En pratique, les développeurs réalisent des *unités intégrables* (un composant logiciel est une unité intégrable) qui sont continûment intégrées dans le prototype en cours. Il est donc possible :

- à chaque intégration d'effectuer des tests de non régression,
- de disposer en cours de développement d'un environnement de test réel (c'est le prototype en cours),
- de contrôler grâce aux scénarios les exigences de haut niveau,
- d'assurer une bonne traçabilité entre les étapes,
- de prévenir à temps les dérapages.

Si des problèmes difficiles sont rencontrés au cours du projet ils peuvent impliquer la réalisation de prototypes dits *de Booch* dont le but est de trouver indépendamment de la ligne générale du projet (afin de ne pas le perturber) les solutions adéquates. Les prototypes de Booch permettent de gérer les risques, ils sont utilisés de telle façon qu'ils puissent assurer une véritable consolidation du plan de développement (il est en effet sans intérêt de définir par exemple un planning si l'ensemble des risques n'est pas maîtrisé). En l'étape n (c'est-à-dire pendant la phase où on réalise le prototype n) une partie de l'équipe de développement résout les problèmes de l'étape suivante n+1 en utilisant des prototypes de Booch menés en parallèle avec l'étape n.

La méthodologie recommande les phases suivantes :

La *phase de conceptualisation* : qui permet de définir le périmètre du système et son contexte. Durant cette phase le projet est découpé en catégories qui peuvent être vues comme des classes de fonctionnalités. On définit aussi l'ensemble des prototypes dont le projet aura besoin.

La *phase de* c*onception de l'architecture* : le but est de définir l'architecture et de spécifier le premier prototype nommé prototype d'architecture (PA). Ce prototype démontrera (on utilise une batterie de tests) non seulement que l'architecture est conforme aux spécifications mais également que tous les risques qui y sont attachés ont été pris en compte.

La *phase d'affinement* : dont chaque étape est la réalisation d'un prototype fonctionnel (PF). Plus précisément, la phase d'affinement est donc caractérisée par la réalisation d'autant de prototypes fonctionnels que nécessaire. Un prototype fonctionnel implémente des fonctions métier contrairement au prototype d'architecture qui est plus technique.

La *phase de validation globale* : qui est en réalité une double activité. Elle comprend une validation proprement dite où l'on montre que chaque exigence a été pris en compte et une "certification" effectuée par une équipe indépendante spécialisée. La "certification" ici est en réalité une validation du produit placé dans un véritable contexte opérationnel (utilisation de bancs spécialisés).

## 1.2 MORSE et MDA (Model Driven Architecture)

Une partielle de réflexion pour mieux nous situer par rapport au MDA. Cette dernière technique stipule que l'on peut converger vers l'implémentation donc le produit final à l'aide de transformations successives de modèles. Comme on le sait, les transformations ne sont pas quelconques, elles respectent le métamodèle dans lequel s'inscrivent les modèles et elles s'appuient en aval sur un modèle de plate-forme.

Face au MDA, la position de MORSE est assez floue. En effet, dans le MDA, le modèle qui permet la génération de code est le PSM (*Platform specific model*). Toutefois pour arriver à ce PSM, il faut dérouler tout un processus qui part du PIM (*Platform independent model*) et qui se termine par l'obtention de ce PSM. Tel que MORSE est décrit, on a l'impression qu'il adresse uniquement la phase PSM → génération de code ce qui paraît très réducteur. Finalement, à partir de quel modèle MORSE active-t-il ? S'il part *strictement* du PIM alors il faut que le projet MORSE intègre quelque part l'ensemble des transformations permettant d'arriver à un PSM. L'une des options qui pourrait être prise est la suivante : le processus de développement d'un projet d'une certaine "largeur" pourrait être composé de plusieurs cycles, chaque cycle comprend une transformation **T** d'un modèle $M_n$ et une vérification **V** (effectuée par MORSE) des propriétés associées à ce dernier. Le dernier cycle est évidemment la génération de code (effectuée aussi par MORSE). Autrement dit, on aurait : $M_{n+1} = \mathbf{V}_n \mathbf{T}_n (M_n)$ avec d'éventuels retours $M_{n+1} \to M_n$. Toutefois, la condition nécessaire pour que de tels cycles tournent est que i) chaque $M_n$ soit vérifiable par MORSE ii) qu'une aide soit apportée à la recherche des transformations **T** (mais a priori, ceci ne fait pas partie de MORSE).

Examinons le premier point i). Le premier modèle $M_1$ est directement sorti de la phase d'élicitation (c'est-à-dire de la phase où modèle et besoin sont changeants). Ce modèle est de haut niveau et bien entendu indépendant de la plate-forme. Puisqu'il s'exprime en des termes très abstraits (hautement conceptuels) sera-t-il traduisible en **L*f*P** ? On peut penser que la réponse est oui car on imagine qu'à ce niveau le contrôle va se résumer uniquement à un agencement de boîtes que l'on saura décrire sommairement. Toutefois, aura-t-on des propriétés fortes et pertinentes à prouver ? Rien n'est moins sûr, il est possible qu'à ce stade on "enfonce des portes ouvertes". Dans ce cas, à quoi sert la translation en **L*f*P** ? Plus

intéressant, que se passe-t-il si on force à ce niveau la génération de code ? Aura-t-on véritablement un exécutable, un simulateur du problème à traiter ? Bref, si on ne génère pas de code alors le passage à **L*f*P** ne sert à rien pour $M_1$ et si MORSE ne propose pas de transformations MDA alors il reste en marge du processus de développement.

Une autre possibilité, serait que $M_1$ soit quasiment le modèle "définitif" (du point de vue de l'architecture du modèle) mais qu'il possède des trous. Par exemple, le modèle $M_1$ comprend trois composants $C_1, C_2, C_3$ structurés par la topologie *T,* ce qu'on écrit $M_1 = T(C_1, C_2, C_3)$ mais l'un deux est vide (il n'est exprimé par exemple que par ses propriétés). Dans ce cas $\mathbf{V}_2(M_2)$ a un sens à condition bien sûr qu'on ait transformé $M_2$ en **L*f*P** ce qui *semble* toujours possible. On pourraît même imaginer qu'au niveau de cette première étape, tous les composants sont uniquement exprimés par leurs propriétés mais alors i) **L*f*P** sera-t-il capable de vérifier quelque chose ii) ne risque-t-on pas encore une fois d'enfoncer à ce niveau "des portes ouvertes" ?

Supposons alors que $M_1$ soit pertinent (plusieurs composants sont plus ou moins détaillés) alors la question qui se pose est : comment obtient-on ce modèle ? Est-il le résultat d'un processus en amont de type MDA ?

On le voit donc la méthodologie associée à MORSE n'est pas évidente et mérite vraiment qu'on s'y attarde.

## 1.3 MORSE et Démarche par prototypage

MORSE doit-il vraiment s'appuyer sur une démarche de type MDA ? En d'autres termes la méthodologie doit-elle préconiser uniquement une suite de transformations de modèles ? La question mérite d'être posée puisque l'alternative existe, c'est la démarche par prototypage : au lieu de transformer, on complète, on ajoute (ce qui n'empêche pas évidemment aussi de transformer). Peut-être faut-il aussi un mélange bien dosé des deux ?

Dans la démarche par prototypage, on va travailler par "strates". On va définir complètement une première strate[1] $S_1$ dotée des propriétés $\varphi_1$, [on écrit : $S_1(\varphi_1)$] c'est aussi le **premier prototype**. Le but de MORSE sera ici de montrer que $S_1$ vérifie bien les propriétés qu'on lui a attribué c'est-à-dire $\varphi_1$, parmi celles-ci il y aura par exemple la propriété `exécutable` vérifiable directement par l'intermédiaire de la génération de code. Evidemment, l'obtention de $S_1$ va nécessiter plusieurs cycles locaux (qu'on détaillera plus tard). Une fois cette première étape franchie, on va ajouter "sur" $S_1$ la deuxième strate $S_2$ dotée des propriétés $\varphi_2$. L'ajout de cette strate va évidemment dans le sens d'une convergence car en réalisant cet ajout on se rapproche quelque part du produit final. Ce qu'il va falloir maintenant que l'on prouve avec MORSE c'est que compte tenu des

[1] $S_0$ est le Runtime.

propriétés précédentes $\varphi_1$ (qui sont vraies puisque vérifiées durant l'étape précédente) alors les propriétés nouvelles $\varphi_2$ sont respectées. D'une certaine manière, il faut montrer $\varphi_1 \Rightarrow \varphi_2$ dans le modèle proposé $M(S_1 \oplus S_2) = M_1 \oplus M_2$ qui représente le **deuxième prototype**. Evidemment, on ne gagnera pas à tous les coups et il est vraisemblable que l'implication en question ne sera pas vraie donc il faudra modifier $S_1$ ou $S_2$. Il y aura donc ici un cycle local à gérer au cours duquel on modifiera séparément les modèles $M_1$ et $M_2$ respectivement associés à $S_1$ et $S_2$ de telle manière que l'on puisse satisfaire les propriétés en question. Ce cycle sera loin d'être facile car il faudra trouver dans $M_1 \oplus M_2$ ce qu'il faut modifier sans casser ce qui est correct (on s'apparente ici à une méthodologie proche de celle du processus B où pour avancer il faut démontrer quelques lemmes indispensables).

Dans cette optique, le processus est construit à partir d'une suite de cycles dans laquelle la progression s'effectue à un moment donné par l'intermédiaire de raffinements effectués principalement sur la dernière strate. En effet, comme on l'a mentionné précédemment, l'ajout de la strate $S_n(\varphi_n)$ implique :

1. Une preuve par MORSE que $\varphi_1 \wedge ... \wedge \varphi_{n-1} \Rightarrow \varphi_n$ ,
2. Une génération de code "partielle" ou bien en considérant comme plate-forme l'ensemble $S_0 \oplus S_1 \oplus ... \oplus S_{n-1}$ ou bien une génération totale capable de générer $S_1 \oplus ... \oplus S_{n-1} \oplus S_n$ sur $S_0$ (c'est-à-dire la *Runtime*).

Ainsi dans cette approche, plus on avance, plus le modèle se complexifie "horizontalement" alors qu'en MDA il se complexifie plutôt "verticalement" (i.e : on raffine - au sens de "on transforme" -).

# 2 Domaine d'application

## 2.1 Objectifs

Du point de vue de la méthodologie de développement de l'application, l'impact de L*f*P est illustré par la Figure 1 . Les parties ``contrôle'' et ``métier'' de l'application sont spécifiés et implémentés de manières différentes:

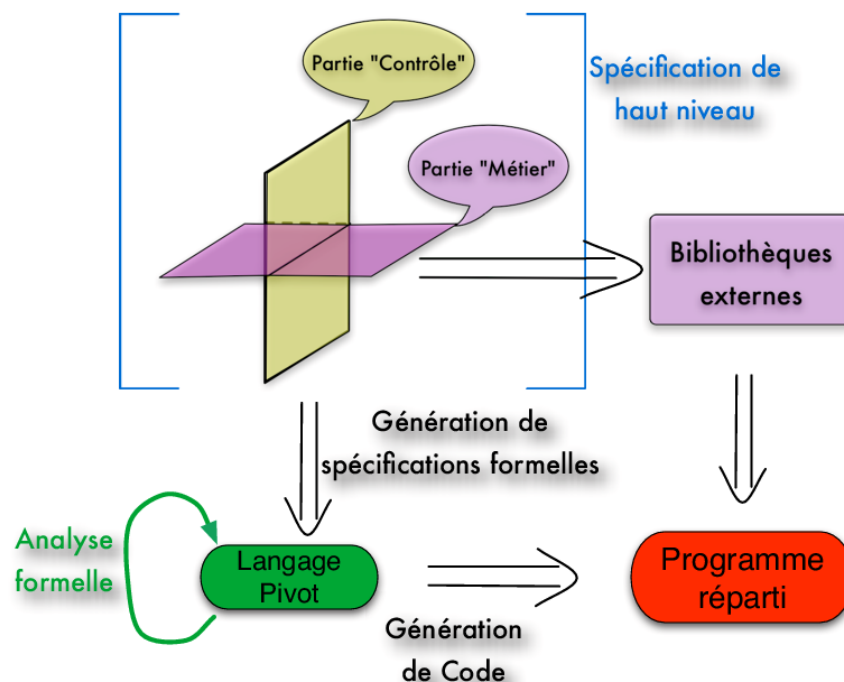


**Figure 1 : Méthodologie de développement avec L*f*P**

L'objectif du projet MORSE est de produire la chaîne de traitement de la partie « contrôle » de l'application répartie. La première étape consiste à la traduire une spécification de haut niveau (UML) en une spécification L*f*P. Une étape de vérification formelle est ensuite appliquée pour s'assurer que la solution retenue respecte les exigences exprimées. Puis le code correspondant est généré.

Le programme réparti ainsi produit utilise les bibliothèques de composants externes pour réaliser les parties ``métier'' de l'application. Le générateur de code produisant le code de ``contrôle'' de l'application ne fait aucune hypothèse sur la manière dont sont produites ces bibliothèques. Les appels aux fonctions de bibliothèque sont spécifiés dans la description du contrôle de l'application sous la forme ``d'appels externes''.

Le langage L*f*P est dédié à la description de la partie contrôle des systèmes distribués. Cela signifie que les spécifications exprimées doivent décrire précisément la manière dont les composants de l'application communiquent et interagissent. La partie

``calculatoire'' de l'application n'est pas modélisable par ce biais. Une liaison entre des composants ``métier'' et la spécification L*f*P doit être établie pour obtenir une représentation globale de l'application.

Les interactions entre les composants ``métier'', et la spécification L*f*P est illustrée sur la Figure 2 ci-dessous. Les classes L*f*P appellent des fonctions appartenant à des bibliothèques prédéfinies. La production de ces bibliothèques n'est pas abordée par le langage L*f*P et sort du cadre du projet MORSE.

Chaque composant métier est constitué d'un ensemble de composants de base interagissant pour remplir une fonction de l'application. Ces interactions doivent respecter un ensemble de contraintes garantissant qu'elles n'influent pas sur la partie contrôle de l'application.

Un composant métier ne doit donc pas :

- Modifier directement l'état courant d'un composant L*f*P, ou une de ses variables,
- Appeler une méthode d'un composant L*f*P,
- Communiquer directement avec une instance de composant manipulée par un autre composant L*f*P.

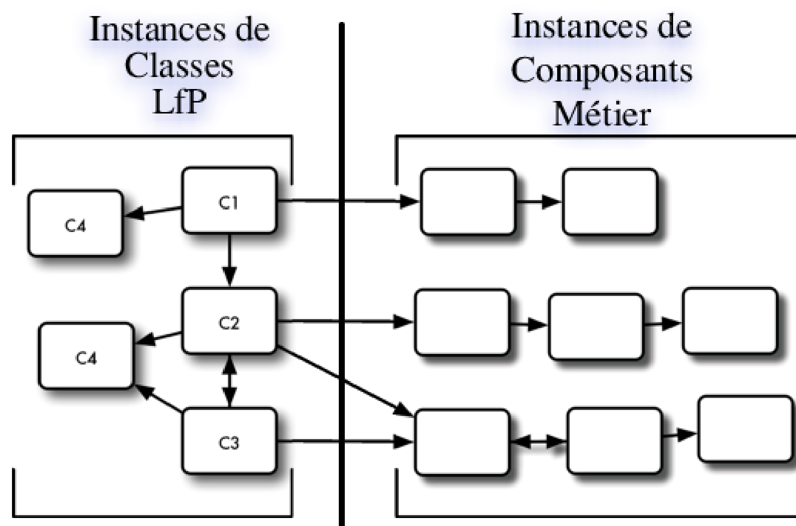


**Figure 2 : Interactions entre les composants L*f*P et les composants métiers**

De même, les composants L*f*P doivent transmettre les messages circulant entre les composants ``métier'' sans altérer leur contenu. L'interaction avec les composants métiers doit être limitée à l'interface prévue par l'implémentation. Si ces contraintes ne sont pas totalement respectées, il sera impossible de vérifier formellement le comportement du système. Un composant de l'application sera donc constitué de l'agrégation d'un composant L*f*P et de l'ensemble des composants métiers avec lesquels il communique directement.

## 2.2 Caractéristiques

## 2.3 Contraintes

# 3 Processus

## 3.1 Vue d'ensemble des étapes du processus

Le processus de développement de la partie « contrôle » de l'application répartie peut être divisé en 2 sous-processus :

- le premier sous-processus débute par une étape de modélisation UML, son but est l'obtention d'un modèle UML « correct », c'est-à-dire d'une part conforme au profil MORSE, et donc propre à être transformé en modèle L*f*P, et d'autre part ayant satisfait les critères des vérifications formelles, appliquées au modèle L*f*P correspondant.
- le second sous-processus part de ce modèle UML « correct » pour aboutir au texte source du programme. Ce texte source est obtenu à partir de la forme L*f*P du modèle, par génération automatique selon des règles de transformation définies.

### 3.1.1 Sous-processus 1 : obtention d'un modèle UML correct

Le schéma de la Figure 3 résume dans sa colonne de droite (sous la forme d'un diagramme d'activités UML) les différentes étapes du sous-processus 1. Chaque étape est représentée sous la forme d'une activité, et les produits en entrée et en sortie d'étape sous forme de flot d'objets.

Ces étapes sont :

- La modélisation UML, au moyen de l'outil Ameos d'AONIX. Cette modélisation se conforme à un profil UML, le « profil MORSE », et répond à des directives spécifiques de modélisation. Le respect du profil UML et des directives garantit que le modèle UML pourra être transformé en un modèle équivalent exprimé en langage L*f*P.
- La génération du modèle L*f*P à partir du modèle UML. Le modèle UML saisi sous Ameos est fourni (sous son format textuel imf) à l'outil ACD d'AONIX. Celui-ci, muni de règles de transformation exprimées dans des « templates », génère le modèle L*f*P correspondant, sous une des formes textuelles de L*f*P (msm ou XML).
- La génération de spécifications formelles du système modélisé. Ces spécifications formelles serviront de données d'entrée pour les vérifications formelles du système.
- La vérification formelle des propriétés du système décrit par les spécifications issues du modèle L*f*P. Cette vérification détecte les failles éventuellement présentes dans le modèle L*f*P, et fournit des indications sur les corrections à apporter à ce modèle.

- Les résultats de la vérification sont ensuite traduits en des termes relatifs au modèle UML. Cette « remontée des erreurs » est nécessaire car la vérification est effectuée sur le modèle L*f*P, et ses résultats ne s'adressent donc pas directement au modélisateur UML. Ce dernier n'est pas censé intervenir sur le modèle L*f*P : après qu'il ait établi le modèle UML, les étapes qui ont suivi ont été réalisées de façon automatique par des outils.
- Si l'étape précédente a détecté des corrections à apporter au modèle UML, le modélisateur intervient en conséquence sur le modèle UML pour le corriger : on revient à l'étape initiale de modélisation UML (puis on suit à nouveau le déroulement des étapes ci-dessus)
- Si au contraire aucune correction du modèle UML n'est préconisée, le modèle UML a atteint un état correct, à partir duquel peut commencer le sous-processus 2, celui de la génération du programme.

Le connecteur **A** représente la fin du sous-processus 1 et le début du sous-processus 2, il indique la liaison entre les schémas des 2 sous-processus (voir Figure 4).

Dans la colonne de gauche de la Figure 3 sont indiqués les documents ou informations nécessaires à la réalisation de certaines étapes, et dans la colonne du milieu, s'il y a lieu, les outils utilisés pour mener à bien certaines étapes.

*documents utilisés* *outils utilisés* *étapes*

modélisation

Ameos-UML2

• Profil MORSE (UML2)
• Directives de modélisation

Modèle UML2
[format Ameos]

Gén. LfP

ACD-UML2

• Règles de transf UML2-LfP
• Templates msm ou XML
pour profil M (ACD2)

Modèle LfP
[format msm ou XML]

Gén. Specs f.

Générateur sp.f.

Specs formelles

Vérification f.

Vérificateur

Résultats des
Vérifs formelles

Remontée des erreurs

Outil E

Corrections à apporter
au modèle UML

[corrections]

[aucune correction]

Modèle UML2
CORRECT

A

**Figure 3 : Sous-processus 1 : obtention d'un modèle UML « correct »**

### 3.1.2 Sous-processus 2 : génération du programme

Le schéma de la Figure 4 résume dans sa colonne de droite (sous la forme d'un diagramme d'activités UML) les différentes étapes du sous-processus 2. Chaque étape est représentée sous la forme d'une activité, et les produits en entrée et en sortie d'étape sous forme de flot d'objets.

Le connecteur **A** indique la liaison avec la fin du sous-processus 1 : le sous-processus 2 reçoit en entrée le modèle UML « correct », ayant passé avec succès l'étape des vérifications formelles.

Les étapes du sous-processus 2 sont :

- La génération du modèle L*f*P à partir du modèle UML « correct ». Cette étape est similaire à la 2ème étape du sous-processus 1, mais on part cette fois d'un modèle UML déjà vérifié avec succès. Le modèle L*f*P est généré sous une des formes textuelles de L*f*P (msm ou XML).
- La transformation du format du modèle L*f*P, en vue de son utilisation en entrée de l'outil ACD d'AONIX : cet outil admet en entrée un format textuel nommé *imf*. Ce langage textuel, conçu à l'origine pour permettre de représenter tout modèle UML, sera étendu pour pouvoir exprimer les notions du langage L*f*P et représenter tout modèle L*f*P (*imf pour LfP*). Un traducteur sera développé pour générer le modèle L*f*P dans ce format à partir de sa forme msm ou XML.
- La génération du programme au moyen de l'outil d'AONIX *ACD-LfP*. Cette version spéciale de l'outil ACD sera développée spécialement pour générer des programmes à partir de modèles L*f*P, selon les mêmes techniques que l'outil ACD existant, qui lui opère à partir de modèles UML. Un métamodèle simplifié de L*f*P sera défini, en tenant compte des besoins particuliers d'ACD. L'outil ACD permet de définir de façon personnalisée les patterns du code à générer. Il analyse le modèle en entrée relativement à un métamodèle. Pour chaque entité (ou groupe d'entités liées) représentant une (des) instance(s) d'entité(s) du métamodèle, l'utilisateur peut spécifier (dans un document appelé « template ») le texte correspondant à générer. Pour une même entité (ou pour un même groupe d'entités), de multiples modes de génération peuvent être définis, le choix entre ces modes étant pilotable par des indications insérées dans le modèle sous forme de stéréotypes, annotations, etc. La génération requiert l'élaboration préalable de 2 documents : l'un définissant les règles de transformation L*f*P-vers-code, l'autre spécifiant à l'usage d'ACD les « templates » de génération qui traduisent l'application de ces règles de transformation en termes d'entités du métamodèle.

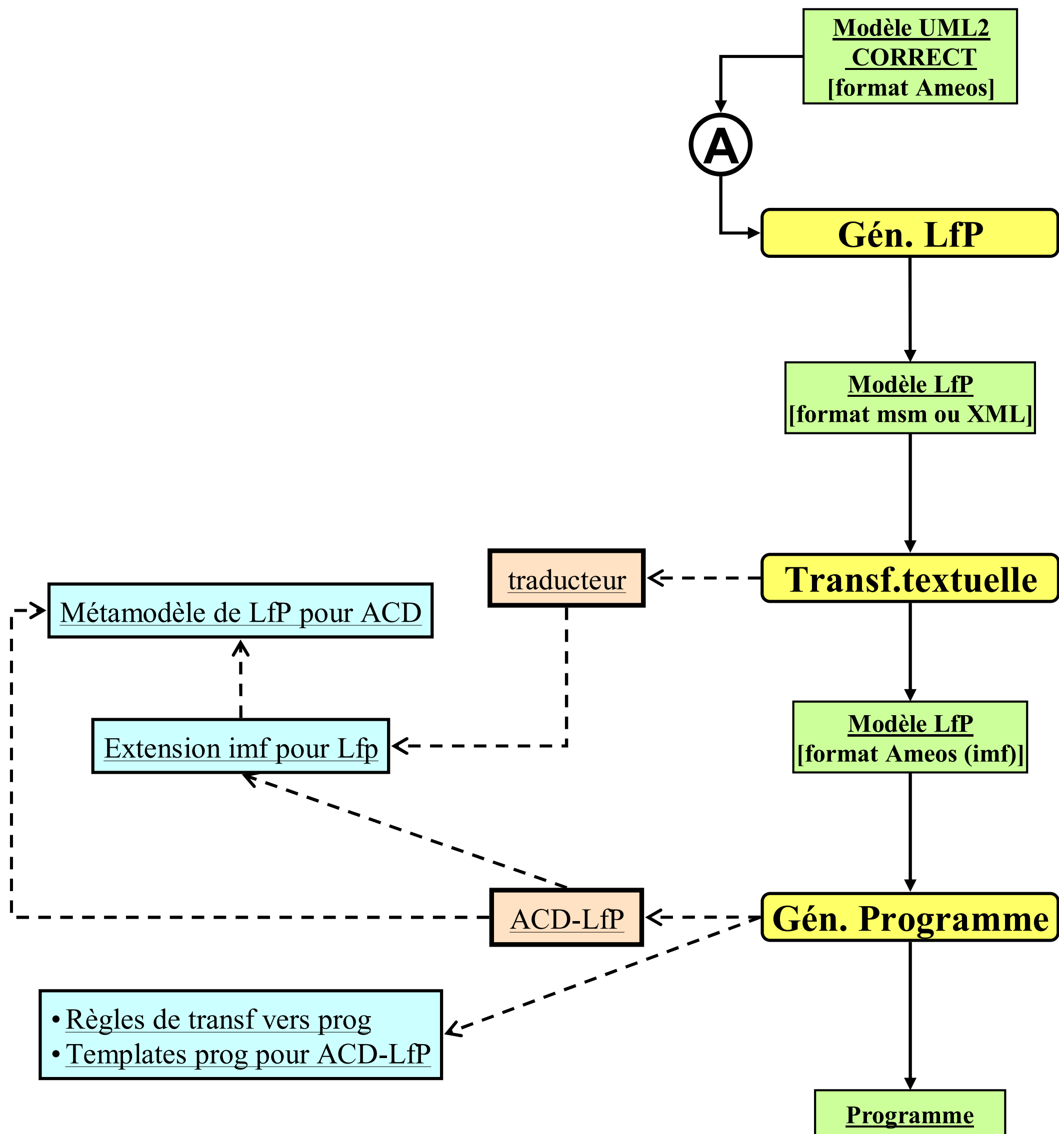


**Figure 4 : Sous-processus 2 : génération du programme**

## 3.2 Ecriture du modèle UML

## 3.3 Traçabilité

## 3.4 Formalisation

### 3.4.1 Comment faire un modèle UML

### 3.4.2 Comment appliquer le profil

### 3.4.3 Comment séparer contrôle et métier

### 3.4.4 Comment faire un cycle de formalisation ?

### 3.4.5 Comment générer le L*f*P

### 3.4.6 Prise en compte des erreurs

### 3.4.7 Comment établir les propriétés à vérifier

### 3.4.8 Vérification

La phase de vérification prend en entrée une liste de propriétés à vérifier et le modèle L*f*P de l'application. La vérification peut être vue comme un procédé itératif de modification du modèle, calcul des états et vérification proprement dite de la propriété.

Cette section décrit successivement l'importance du choix du modèle, la manière dont sont utilisés les DDD (diagrammes de décision de données) dans le processus de vérification et enfin le point de vue de l'utilisateur.

#### 3.4.8.1 Analyse préliminaire

La première étape confiée au concepteur consiste à analyser les propriétés à vérifier et à décider, pour chacune d'elles, de dériver du modèle initial un modèle plus approprié à la vérification de la propriété en question. Le choix se porte en général vers un modèle plus simple ou plus petit, mais équivalent au modèle initial du point de vue de la propriété analysée.

Une analyse de complexité sur l'application et sur le modèle peut donner des indications sur les simplifications à appliquer au modèle. Cette analyse est à la charge du concepteur.

En revanche, le processus de modélisation doit prendre en compte les problèmes posés par la vérification et faciliter les dérivations du modèle. Ces dérivations seront d'autant plus faciles si le modèle satisfait des critères qualitatifs tels que paramétrisation, structuration, symétries ...

Cette analyse préliminaire fait partie du processus itératif dans la mesure où il est difficile de prévoir à l'avance si l'espace des états peut tenir en mémoire pour un modèle donné.

#### 3.4.8.2 Utilisation des DDD

Les DDD sont utilisés pour représenter l'ensemble des états possibles du système. Dans le cadre du projet MORSE, les aspects relatifs aux DDD ne sont pas visibles par l'utilisateur, mais sont traduits dans un format cohérent avec le modèle à vérifier.

La représentation compacte des données, l'efficacité du calcul des états les rendent particulièrement adaptés. Les préconditions et le franchissement des transitions du système sont représentés par des homomorphismes inductifs qui sont le mécanisme interne des DDD pour le calcul des états successeurs. L'application des homomorphismes sur un état initial du système permet de calculer toutes les évolutions possibles du système. Un algorithme de calcul des états permet de contrôler l'application des homomorphismes et d'accumuler les états du système.

Les propriétés du système sont aussi représentées au moyen d'homomorphismes inductifs. Une fois que les états sont calculés, la vérification proprement dite consiste à appliquer les homomorphismes représentant les propriétés à vérifier sur l'espace des états.

#### 3.4.8.3 Le processus de vérification vu de l'utilisateur

La Figure 5 ci-dessous résume le processus de vérification et répartit les tâches entre l'utilisateur et l'outil.

Durant le processus de vérification, le concepteur décrit les propriétés à vérifier dans un langage qui devra être défini. Ces propriétés sont ensuite traduites en homomorphismes qui sont cachés de l'utilisateur.

Le système décrit en L*f*P est traduit en composition d'homomorphismes représentant les mécanismes de base de L*f*P et paramétrés par les données du modèle. Le codage de l'état est aussi généré automatiquement et abouti à la génération d'un DDD initial après une phase d'élaboration. Le codage de l'état ainsi que les homomorphismes générés sont cachés de l'utilisateur. Un moniteur, paramétré par un algorithme de calcul des états, est configuré et lancé. L'algorithme de calcul des états est aussi paramétrable par l'utilisateur.

Le calcul des états se fait en appliquant itérativement les homomorphismes sur l'ensemble des états courants en suivant le schéma décrit par l'algorithme de calcul.

Les homomorphismes représentant les propriétés à analyser sont appliqués sur l'espace des états obtenu.

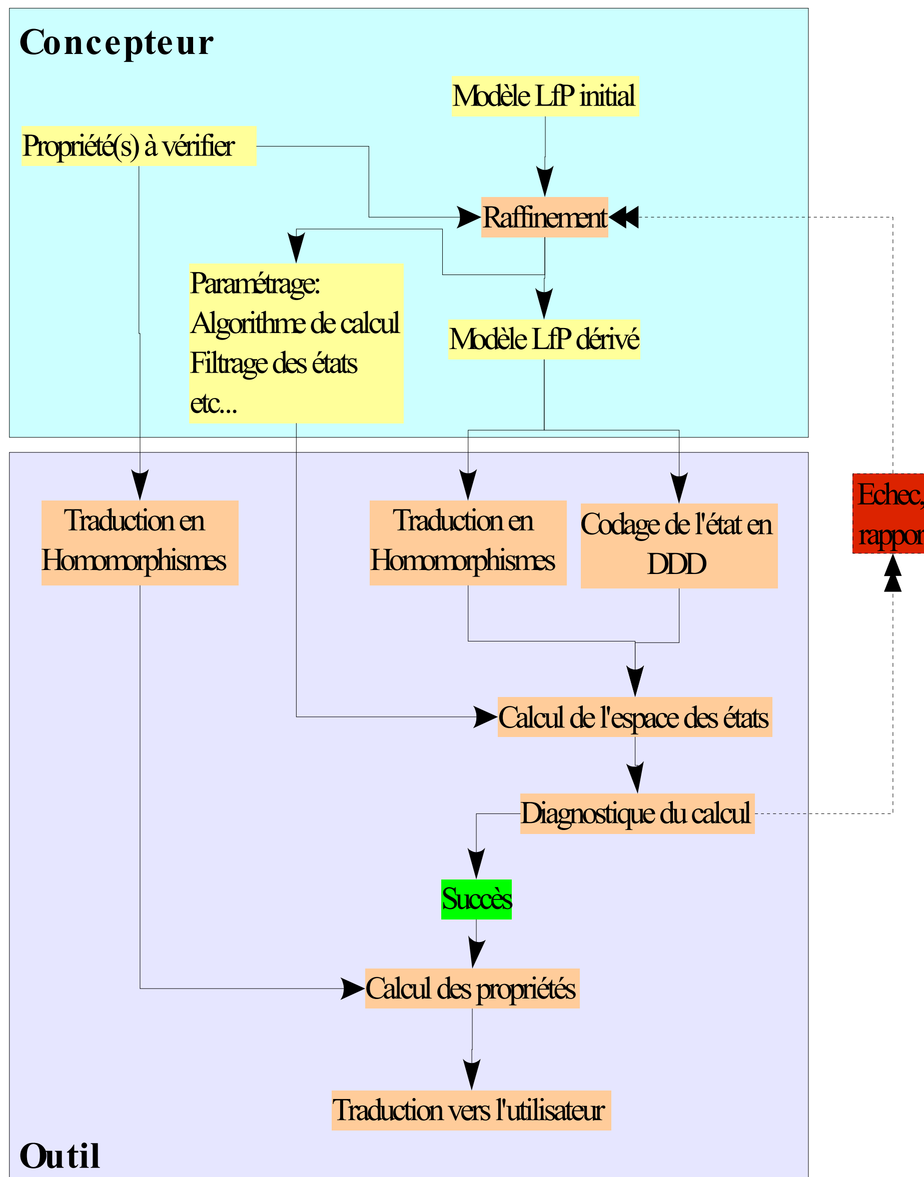


**Figure 5 : Répartition des tâches entre le concepteur et l'outil, organigramme général**

Des informations capturées durant le calcul des états doivent permettre à l'utilisateur de faire des choix sur le modèle lorsque l'espace des états est trop grand. Ces informations sont représentées sur la figure par la flèche de retour en cas d'échec du calcul des états.

Pour chaque propriété, le retour des résultats à l'utilisateur se fait sous la forme d'un ensemble d'états retranscrit dans un format lisible cohérent avec le modèle. Parmi les propriétés, on peut distinguer les propriétés générales de non blocage, de couverture du modèle, etc. ..., d'autres propriétés spécifiques au système. Les propriétés générales peuvent être directement traitées par l'outil sur demande de l'utilisateur.

#### 3.4.9 Génération de code (composants métier)

### 3.5 Directives de génération de code, du modèle…

### 3.6 Confrontation pratiques en cours (se positionner par rapport…)

# 4 Traitement de grosses applications (passage à l'échelle)

# 5 Glossaire

Définition de la formalisation, composant, vérification, modèle,…

# 6 Abréviations et acronymes

UML Unified Modelling Language

L*f*P A compléter…

# 7 Références

[ANN-TECH] Projet MORSE, ANNEXE Technique.

[FG-syntax] Frédéric Gilliers, BNF de la grammaire des attributs des diagrammes L*f*P, 12/06/2003.

[FG-sema] Frédéric Gilliers, Description de la sémantique du langage L*f*P, 12/06/2003.

[MAS-031114] Maurice Assouline, Réflexion technique pour la transformation de UML vers L*f*P, 14/11/03.

[MAS-040621] Maurice Assouline, Représentation en UML des éléments de modélisation L*f*P, 21/06/04.