Document d'orientation "interface"

(Spécification de conception)

Diffusion : Voir dernière page

Qualité	Pour les rédacteurs	Pour les vérificateurs	Approbateur
Fonction	Resp. Interface	Resp. Qualité	Resp. Projet
Nom	M. Lazareff	A.M. Vuillot	L.Cambier
Visa			

Support informatique : GCL ELSA

Date de mise en application : Immédiate

HISTORIQUE		
VERSION ÉDITION	DATE	CAUSE et/ou NATURE de l'ÉVOLUTION
1.0	30 juin 2000	Évolution dans le choix des langages

TABLE DES MATIÈRES

1. INTRODUCTION 5

2. PRINCIPES GÉNÉRAUX 7

2.1. L'IHM comme cadre sécurisé des actions de l'utilisateur 7

2.2. Niveaux d'expertise 7

2.3. Langage de description 8

2.3.1. Fichiers de données 8

2.3.2. Rangement, hiérarchisation 8

2.3.3. Objets et langage de description 8

2.4. Compromis syntaxe/généralité du langage d'interface 8

2.5. Langage de haut niveau 9

2.6. Le cas des interfaces graphiques 9

3. QUELS LANGAGES POUR *ELSA* 11

3.1. Mini-langage MUSE–*elsA* 12

3.2. Langage Python–*elsA* 12

3.3. Langage Python–*elsA*_API 12

3.4. Interface et langage graphique PyG*elsA* 13

3.4.1. Efficacité 13

3.4.2. Sûreté 13

4. STOCKAGE, PERSISTANCE ET RÉUTILISATION 15

4.1. Fichiers de scripts 15

4.2. Objets 15

4.3. Contextes 15

5. PRÉ- ET POST-TRAITEMENT 17

6. PERSPECTIVES 19

DIFFUSION TYPE 21

Page sans texte

1. INTRODUCTION

L'interface d'***elsA*** comporte actuellement deux langages accessibles aux utilisateurs, MUSE–*elsA* et Python–*elsA*. Dans l'optique de la suppression éventuelle de MUSE–*elsA*, les avantages et inconvénients de ces deux langages sont comparés. L'apparition de l'interface graphique **PyG*elsA*** devrait jouer dans le sens d'un plus grand confort d'utilisation, tout en permettant l'utilisation directe du langage Python–*elsA* (dans une console en mode texte).

Les perspectives d'évolution de l'interface utilisateur d'***elsA*** sont analysées à partir d'un retour aux concepts de base de l'interface. Cette évolution doit tenir compte de la variabilité des compétences des utilisateurs, aussi bien d'un utilisateur à l'autre que d'un aspect à l'autre du logiciel pour un même utilisateur, *cf.* 2.2.

Page sans texte

2. PRINCIPES GÉNÉRAUX

2.1. L'IHM comme cadre sécurisé des actions de l'utilisateur

La notion d'IHM (Interface Homme–Machine) suppose une séparation entre la fonction calculatoire (ou plus généralement exécutoire) du logiciel et la spécification (par l'acteur humain) de la tâche à réaliser.

Cette séparation est également présente dans le concept d'API (Application Programming Interface), à un niveau plus proche de la machine.

En prenant la définition d'un cas de calcul de façon destructive (ou soustractive) plutôt que constructive (ou additive), on peut dire que le noyau de calcul réalise une première restriction des états possibles de l'ordinateur, en établissant des liens entre eux. L'API pousse cette restriction un peu plus loin, à la fois statiquement (par des contrôles) et dynamiquement (en masquant certaines programmations possibles dans la bibliothèque).

L'IHM est la dernière strate de cette limitation des degrés de liberté de l'ordinateur. Elle doit permettre à chaque utilisateur de réaliser les tâches prévues, éventuellement modulables (limitables) en fonction de sa compétence ou du contexte de travail.

On s'intéresse ici principalement à l'utilisation interactive, pour laquelle les mécanismes de protection (limitation des actions) et de présentation des possibilités sont potentiellement plus efficaces, et les développements à réaliser les plus importants. Le traitement détaché « par lots » ("batch") est toujours possible, en particulier à partir du fichier de trace d'une session interactive, *cf.* 4.

À terme, la distinction interactif/batch pourrait s'amenuiser, par introduction d'un mécanisme de pilotage dynamique des tâches en cours. Dans cette optique, un calcul lancé en interactif pourrait être *détaché* en batch, et inversement l'utilisateur pourrait désigner (à travers un interface de gestion, éventuellement séparé de l'interface de création) un calcul batch, pour analyser sa convergence ou modifier ses paramètres dans l'interface, utilisé comme éditeur.

2.2. Niveaux d'expertise

Par rapport à l'API, l'IHM se place à un niveau de compétence moins élevé. Cependant, il doit fournir aux utilisateurs chevronnés des fonctionnalités avancées pour le traitement efficace de cas de calcul complexes.

Un utilisateur n'est pas nécessairement novice ou expert de façon uniforme, et passe fréquemment d'un rôle à l'autre suivant les fonctionnalités qu'il utilise.

2.3. Langage de description

La *description* d'un cas de calcul peut être réalisée de façon plus expressive que par la simple énumération d'un nombre plus ou moins grand de paramètres, voir ci-dessous différentes étapes de classification/formalisation/abstraction.

2.3.1. Fichiers de données

Les logiciels d'ancienne génération se contentaient généralement de fichiers de données, spécifiant de façon plus ou moins ordonnée les paramètres (dont le nom d'un fichier séparé de maillage) nécessaires au calcul, avec un rangement rigide lié par exemple à un format FORTRAN. L'introduction de nouveaux paramètres rend difficilement évitable une perturbation du rangement.

Cette solution d'interface garde l'avantage de la simplicité, mais est rapidement limitée du point de vue conceptuel par une description de trop bas niveau (manque d'abstraction).

2.3.2. Rangement, hiérarchisation

Une première étape au-delà de la simple énumération est du niveau de l'« épicerie », et consiste en un rangement aussi précis que possible des paramètres dans des « boîtes » (éventuellement imbriquées de façon hiérarchique) qui donnent une première idée de leur relations et importances relatives.

Cette articulation des paramètres les uns avec les autres permet une économie de pensée considérable, en autorisant la suppression immédiate de branches entières de l'arbre de leurs dépendances (si un tel arbre existe, au moins par morceaux) par un seul trait de « scie » logique.

2.3.3. Objets et langage de description

Une seconde étape est de munir les « boîtes » de paramètres d'une structure d'objet, qui permet de les manipuler en-dehors du contexte initial d'un fichier de données. Chacun de ces objets décrit alors une facette du problème, et peut être réutilisé pour un autre problème plus sûrement qu'un bloc de lignes copié entre deux fichiers de données. La structure d'objet peut même lui permettre (grâce au mécanisme générique d'héritage) de s'adapter à un nouvel environnement sans avoir recours à des clés logiques dans la description du problème.

Le langage de description (ou « de commande ») apparaît alors naturellement comme la combinaison d'un certain nombre de méthodes génériques (existant pour toutes les classes d'objets) et des méthodes propres à chaque classe.

2.4. Compromis syntaxe/généralité du langage d'interface

Le langage d'interface peut se limiter à quelques opérations de base, auquel cas la syntaxe peut être réduite à presque rien. Cependant, sans tomber dans l'excès, quelques symboles réservés aident

grandement à exprimer des structures évoluées. Ce compromis est résolu différemment pour les langages MUSE–*elsA* (dont le seul type évolué est la liste) et Python–*elsA* (qui utilise les types de Python, dont la liste et le dictionnaire).

2.5. Langage de haut niveau

Sur la base du langage de description élémentaire Python–*elsA*, il est possible de développer des fonctionnalités de plus haut niveau, concernant des abstractions de l'ordre de la *configuration* de calcul et non plus du paramètre ou du cas de calcul. Par exemple, on peut envisager une classe de configurations `helico`, dans le contexte de laquelle les versions dérivées des classes de description élémentaires (`model`, `numerics` ...) acquièrent de nouvelles propriétés spécifiques aux hélicoptères.

2.6. Le cas des interfaces graphiques

L'interface graphique en développement constitue une simple surcouche de l'interface textuel. Les concepts sont les mêmes dans les deux cas, seule la représentation change. Les `widgets` sont des métaphores graphiques pour les éléments du langage de description.

Un avantage de l'interface graphique est une plus grande densité d'information (du logiciel vers l'utilisateur), grâce à l'utilisation des modes graphiques « haute résolution » (SVGA=1024x768 et plus) des moniteurs, ainsi que la possibilité de *manipulation directe* des objets [1].

1. Par exemple « glisser-déplacer » (en anglais `drag and drop`)

Page sans texte

3. QUELS LANGAGES POUR *elsA*

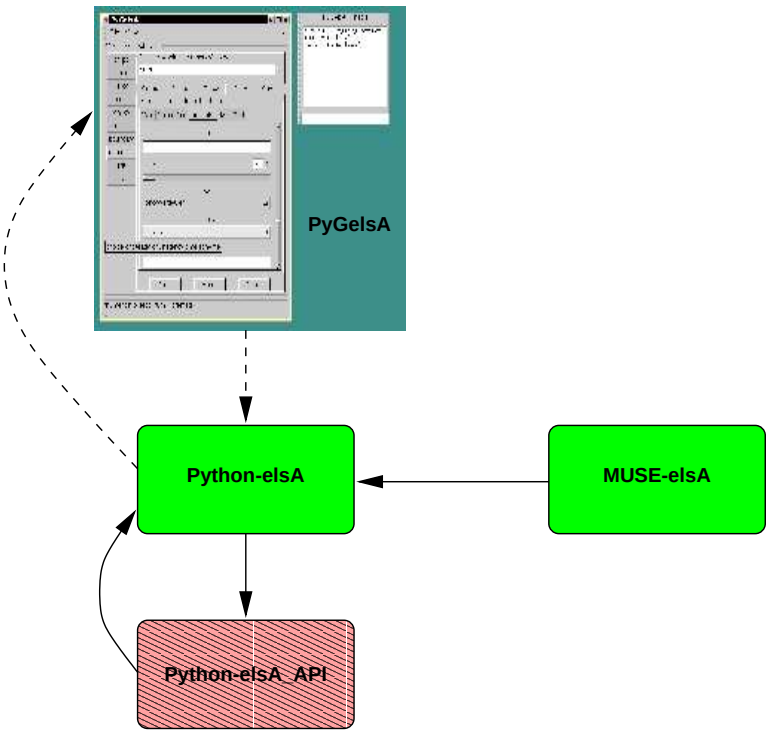

Fig. 3.1 – Relations entre les langages de commande d' ***elsA***

elsA est actuellement interfacé avec deux langages « utilisateur », auxquels s'ajoute Python–*elsA*_API, réservé aux développeurs :

MUSE–*elsA*
```
create numerics "num1"
define numerics "num1" cfl 10.
```
Python–*elsA*
```
num1 = numerics()
num1.cfl = 10.
```
Python–*elsA*_API
```
num1 = DesNumerics('num1')
num1.getTimeInteg().setF('cfl',10.)
```

Ces langages peuvent être utilisés séparément ou conjointement dans un script.

Le fichier de trace automatiquement généré à l'exécution contient la version Python–*elsA* du script (et donc la traduction des éventuelles commandes MUSE–*elsA*). Le fichier de débogage, généré si l'option `-D` est spécifiée, en est la version Python–*elsA*_API.

L'option de ligne de commande `--api2user` permet de plus de regrouper les atomes en macro-attributs et d'optimiser les scripts Python–*elsA* et Python–*elsA*_API.

3.1. Mini-langage MUSE–*elsA*

Le mini-langage MUSE–*elsA* est destiné à faciliter la transition, depuis les précédents langages MUSE–FLU3M et MUSE–CANARI, vers un langage de description pleinement objet. Il ne comporte volontairement pas de fonctionnalités évoluées, afin de conserver une syntaxe aussi simple que possible.

Ce mini-langage est plus rigide que Python–*elsA* (voir ci-dessous), car la définition des valeurs d'attributs doit (dans certains cas) être réalisée selon des groupements prédéfinis, ex :
`define boundary "bnd1" type join jtopo nonperiodic jtype match`
Ces groupements facilitent la définition du cas de calcul pour les utilisateurs novices (en évitant les oublis), et rendent le script plus compact (moins de lignes).

3.2. Langage Python–*elsA*

Python–*elsA* est une extension du langage Python, bâtie sur l'API Python d' ***elsA*** . Elle fournit des fonctionnalités de contrôle (domaine de définition, cohérence interne et externe des objets) adaptées à l'usage direct par l'utilisateur.

L'information est manipulée sous forme de paires attribut/valeur, avec peu de méthodes : constructeurs, accesseurs `set` et `get`, méthodes `show`, `submit` ....

Le rangement et la hiérarchisation utilisent quatre niveaux (classe, sous-classe, macro-attribut [1], attribut) dont deux (sous-classe, macro-attribut) sont optionnels.

Des instructions de boucle implicite (`sequence`) et de définition de dictionnaire de valeurs (`setDict`) permettent de compacter considérablement les données répétitives (voir index du Manuel Utilisateur). Les macro-attributs constituent (par rapport à la référence individuelle de leurs atomes) une autre forme de compactage.

L'extensibilité est fournie en particulier par création de nouvelles classes (ex. `helico`, *cf.* 2.5) et par modification (dans le module `EpUserDefs` destiné au paramétrage par l'utilisateur) de l'héritage des classes de description existantes.

3.3. Langage Python–*elsA*_API

Ce langage [2] correspond à l'utilisation directe de l'API Python d' ***elsA*** et ne sert qu'au débogage ou au développement. Il est peu lisible (relativement à Python–*elsA*) mais complètement explicite (pas d'utilisation de concepts de haut niveau). Il ne comporte pas les méthodes de vérification de Python–*elsA*, mais est automatiquement converti dans ce langage par l'interface utilisateur.

L'utilisation d'un script Python–*elsA*_API à travers l'interface (et non directement en entrée de l'interpréteur `elsA.x`) est donc soumise aux mêmes vérifications que si le langage était Python–*elsA*.

1. Les macro-attributs sont des attributs à valeur de liste, définis par la liste de leurs atomes. Il est ainsi possible de référencer nommément aussi bien la liste de valeurs que ses éléments.
2. Qui n'est pas documenté au niveau utilisateur mais seulement dans le Developer's Guide.

3.4. Interface et langage graphique PyG*elsA*

L'interface graphique **PyG*elsA*** fournit, par l'intermédiaire d'une console Python intégrée, une implémentation cohérente de la version textuelle du langage Python–*elsA* et de sa représentation graphique.

À chaque attribut atomique correspond un `widget`. Les sous-classes de description (masquées en Python–*elsA* et MUSE–*elsA*) sont représentées par des pages de carnets à onglets, et les atomes de macro-attributs (ex. :`dim = [dim1, dim2, dim3]`) sont regroupés dans des cadres libellés au nom du macro-attribut (voir le Manuel Utilisateur). Certains boutons sont dupliqués à plusieurs niveaux, leur action dépend alors du contexte correspondant, *cf.* 4.3.

Tous les concepts évolués inspirés par l'interface graphique sont codés dans et utilisables depuis la couche « texte » de l'interface. La différence est dans la commodité d'utilisation (ex. : valeurs par défaut au niveau du sous-objet accessibles en mode graphique par cliquage sur le bouton local `ElsaDefs` au lieu de la commande en mode texte : `use_subc_ElsaDefs('num1','TimeInteg')`).

3.4.1. Efficacité

Son efficacité (pour la partie graphique, hors console Python), provient :

- pour les utilisateurs novices, de l'explicitation de la structure fournie par les menus déroulants et autres structures graphiques ;
- pour les utilisateurs expérimentés, d'une adéquation éventuelle [3] entre la représentation visuelle des concepts du langage et leur image mentale. La mémoire visuelle (la plus efficace pour la plupart des individus) est alors mise à contribution pour le déroulement des tâches, tant en projection vers le futur qu'en retour en arrière, et diminue la fatigue dans le cas de tâches complexes. Ceci suppose un arrangement logique des différents éléments graphiques, et le masquage (grisé) de ceux qui sont inactifs.

3.4.2. Sûreté

Grâce à l'interactivité plus grande, l'interface graphique donne accès de façon sûre (rétroaction visuelle) à des mécanismes qui peuvent s'avérer dangereux en mode texte. Il est par exemple possible de visualiser directement l'effet de la requête « valeurs par défaut ». Le mode texte permet presque le même mécanisme, mais avec une distanciation qui ne favorise pas les réflexes de vérification /correction.

De plus, il est naturel dans l'interface graphique que le résultat des actions dépende du contexte *cf.* 4.3, alors que ce même contexte doit généralement être explicité dans l'interface textuel.

Remarque : La vérification de cohérence dans l'interface graphique dépend en grande partie de l'accès à l'arbre des dépendances entre les attributs. Cet arbre peut être partagé avec le noyau C++, pour lequel il est un élément implicite ou explicite de la `factory`.

3. Si l'interface est bien conçu ...

Page sans texte

4. STOCKAGE, PERSISTANCE ET RÉUTILISATION

On a envisagé ci-dessus la définition d'un cas de calcul et les langages de description aptes à cette définition. Plus généralement, on considère maintenant l'interface comme un éditeur de cas de calcul, permettant la réutilisation.

Le fichier texte n'est pas le seul moyen de stockage du cas de calcul. Après un rappel de la méthode de travail habituelle (fichiers de script), on envisage d'autres mécanismes de stockage et de réutilisation.

4.1. Fichiers de scripts

La forme la plus simple de définition d'un cas de calcul est le *script*, un programme écrit dans le langage de description qui contient les instructions nécessaire à la création et à l'utilisation des objets de description correspondant au cas de calcul.

Classiquement, le stockage du script utilise directement le système de fichiers du système d'exploitation, mais pourrait passer par une base de données.

La réutilisation de fichiers de script passe par des opérations de copier/coller, méthode simple et efficace dans la plupart des cas mais peu sécurisante.

4.2. Objets

Plutôt que de stocker le programme de création des objets, il est possible de stocker les objets eux-mêmes. Il reste alors à définir le programme complémentaire pour leur utilisation (équivalent des appels aux méthodes `submit` ...).

Dans le cas du mécanisme de persistance fourni par Python (module `pickle`), il est possible d'utiliser la surcharge des classes pour adapter le comportement des objets au moment de leur dé-stockage[1].

4.3. Contextes

Le contexte est une abstraction, qui englobe tous les conteneurs d'information de l'interface (session de calcul, cas de calcul, objet ou sous-objet de description, macro-attribut). Dans l'implémentation envisagée, les contextes partagent des méthodes de stockage et de validation (hérités de la classe `EpContext`) qui les rendent équivalents du point de vue de l'utilisation, quel que soit par ailleurs le niveau de l'objet dans la hiérarchie.

Le stockage des contextes permet de les importer (et éventuellement de les modifier) pour être incorporés à un cas de calcul au cours d'une session. L'ensemble des contextes accessibles, stocké

1. Le stockage par sérialisation des objets de description n'est pas actuellement implémentée (difficulté liée à l'héritage des classes C++).

sur une base de données, est une version plus générale (et qui l'englobe) de la solution courante de modification de cas de calcul existants choisis dans la base de cas-tests.

Remarque : La notion de contexte dans l'interface graphique est la même qu'en langage naturel, à la différence que son domaine est défini sur l'écran de visualisation (position de la souris ...) et non dans le temps du discours (et qu'il n'y a pas d'ambiguité !). L'interface textuel doit lui définir explicitement le contexte de chaque action (voir l'exemple des valeurs par défaut en 3.4).

5. PRÉ- ET POST-TRAITEMENT

Les fichiers de maillage peuvent être notablement améliorés par l'adoption d'un format incluant les éléments topologiques. Le format CGNS en particulier permet la gestion d'un arbre de données adéquat pour les maillages et les résultats (mais ne propose pas de solution générale pour les paramètres de calcul et les méthodes de vérification de cohérence).

Dans le cas de configurations complexes, la définition de la topologie constitue l'essentiel des fichiers de script, alors que les objets de description de maillage pourraient être réduits à la définition du nom du fichier (ou à une requête de base de données). Cette évolution dépend de l'acceptation large d'un tel format par les partenaires industriels.

Remarque : La spécification actuelle d'interface d' ***elsA*** ne contient, pour le pré- et le post-traitement, que des éléments relatifs au choix des variables et au format (VOIR3D, TECPLOT) et type (ASCII, binaire) des fichiers d'entrée ou de sortie. Le détail des opérations de traitement n'est pas spécifié (ex: macro TECPLOT), et les résultats de calcul au niveau final ne sont pas des objets.

Page sans texte

6. PERSPECTIVES

Le mini-langage MUSE–*elsA* est conservé dans une phase transitoire, jusqu'à ce que l'usage du langage Python–*elsA* (éventuellement à travers l'interface graphique **PyG*elsA***) se généralise auprès des utilisateurs. En particulier, l'introduction d'une vérification dynamique de cohérence et de complétude (changement de l'aspect de l'interface indiquant les dépendances) est une alternative au regroupement de définitions d'attributs utilisé par MUSE–*elsA* pour imposer une introduction cohérente des données. Par ailleurs, les utilisateurs de fichiers de script volumineux devraient être les premiers à réaliser l'avantage de Python–*elsA* (et de ses fonctionnalités évoluées, basées sur le langage Python) sur MUSE–*elsA*, d'autant que la traduction est automatique.

Le langage « de commande » sera Python–*elsA*, y-compris pour les développeurs (qui utilisent actuellement directement l'API). Ceci est grandement facilité par l'existence du fichier de configuration `EpAttrDefs.py`, qui peut aisément être édité pour introduire de nouveaux attributs ou classes de description, contrairement à MUSE–*elsA* qui nécessite une intervention relativement complexe.

L'interface graphique **PyG*elsA*** devrait devenir d'utilisation courante, grâce à l'intégration des modes texte et graphique qui le rend efficace aussi bien pour les utilisateurs novices que pour les développeurs. La plus grande densité d'information fournie par le mode graphique facilite la manipulation de concepts plus évolués (naturels en langage parlé), comme les contextes.

La conception de **PyG*elsA*** prévoit (par son insertion dans l'environnement GNOME) une interaction par glisser-déplacer entre ***elsA*** et les logiciels graphiques compatibles avec le standard CORBA (ou qui peuvent être emballés par une couche CORBA)[1]. Ce développement n'est pas envisagé à court terme, et son intérêt dépend étroitement du type d'exploitation du logiciel.

Le changement de mode de travail impliqué par ces évolutions du « langage » de commande (textuel ou graphique) devra être validé dans le cadre d'une appropriation par l'équipe ***elsA*** et par des utilisateurs pilotes.

1. Une interaction CORBA est également possible au niveau Python ou C++.

Page sans texte

DIFFUSION TYPE

Archives Secrétariat

Rédacteurs

Développeurs ***elsA***

Utilisateurs ***elsA***

Responsable Documentation

FIN de LISTE