Interfaçage du code système LiCore du Réacteur MSFR

Rapport de Stage
2017/2018

Florian PASSELAIGUE
Etudiant en Génie Energétique et Nucléaire (2ème année)

Maîtres de stage :

LPSC
Prof. Elsa MERLE
merle@lpsc.in2p3.fr

Corys
Mme Béatrice PLATEL
bzoppe@corys.fr

Remerciements

J'aimerais remercier les personnes qui ont rendu ce stage possible et en ont fait une expérience très enrichissante :

- Elsa Merle, pour m'avoir donné l'envie de découvrir le MSFR et l'opportunité d'y apporter une contribution durant ce stage

- Les membres de la R&D Corys, toujours disponibles pour me sortir des méandres de l'informatique. Notamment Olivier Bruneau, grâce à qui ce projet a pu être mené à son terme, ainsi que Béatrice Platel ma tutrice de stage.

- L'équipe MSFR au complet, pour son accueil chaleureux.

Laboratoire de Physique Subatomique et Cosmologie

Les domaines de recherche du Laboratoire de Physique Subatomique (LPSC) de Grenoble recouvrent de nombreux domaines, de la physique la plus fondamentale (particules, quarks, gluons) à la cosmologie en passant par les secteurs de l'énergie et de la santé. Au sein de ce dernier axe, la thématique Nucléaire Energie est répartie sur les différents aspects de l'avenir du nucléaire via les deux équipe Physique des Réacteurs et MSFR.

Un de ces objectifs est l'amélioration des réacteurs utilisés actuellement afin d'assurer une bonne transition vers la prochaine génération de technologie nucléaire. D'autre part les réacteurs pilotés par accélérateur pourraient permettre de clore le cycle combustible des centrales en fonctionnement. Enfin, l'équipe MSFR travaille au développement de la génération IV de réacteurs nucléaires, avec en ligne de mire un concept de réacteur à combustible liquide : le *Molten Salt Fast Reactor* sur lequel porte ce stage.

Corys

Corys est une entreprise qui développe, installe et maintien des simulateurs informatiques. Son activité se développe sur trois domaines. Le département Energie s'occupe à la fois de simulateurs destinés à l'ingénierie (thermohydraulique, nucléaire, réseaux électriques) et à l'exploitation d'hydrocarbures (simulation de production pétrochimiques, etc…). D'autre part les produits développés par le département Transport sont utilisés pour la formation de conducteurs de train/tramway.

Tous les différents simulateurs du département Energie, destinés au nucléaire notamment, sont conçus et exploités sur la plateforme ALICES® qui permet une gestion modulaire des différents produits et qui a été utilisée lors de ce stage.

Sommaire

Glossaire

- **MSFR** : « Molten Salt Reactor », concept de réacteur de génération IV développé au LPSC
- **ALICES®** : Plateforme de développement de simulateurs développée par Corys
- **IHM** : Interface Humain-Machine : désigne l'interface graphique d'un programme
- **CPZ** : « Cinétique Point par Zone » : modèle neutronique du code système LiCore
- **Classe** : En programmation, désigne l'ensemble des attributs et méthodes d'un objet
- **Hérédité** : Principe par lequel une classe reçoit les attributs et méthodes d'une classe mère
- **Objet Nv1**: Brique élémentaire des simulateurs Alices® effectuant les calculs
- **Prototype** : Equivalent de l'objet Nv1 pour les IHM
- **Document** : Représentation d'un sous-système composé d'objets Nv1
- **Image** : Equivalent du document pour les IHM
- **Module Nv3** : Module Alices® construit directement, sans utiliser ni objet ni document
- **Configuration :** Description du simulateur incluant documents Nv2 et modules Nv3
- **JNI :** Bibliothèque C++ permettant de faire appel à du code Java

Liste des figures et tableaux

Introduction

Dans le cadre d'une transition énergétique durable, il est capital de pouvoir disposer d'une source d'énergie qui soit abondante et aussi neutre que possible vis-à-vis de l'environnement, notamment matière de gaz à effet de serre. C'est cet objectif que s'est donné le Forum Génération IV, une initiative internationale visant à cadrer le développement d'une nouvelle génération de systèmes nucléaires.

Dans cette optique, différents critères ont été définis sur la durabilité, la sécurité, la sureté et l'aspect économique, sans oublier la résistance à la prolifération d'armes nucléaires. Six concepts de réacteur répondant à ces critères, parmi lesquels le MSFR, ont été retenus. L'équipe travaillant sur ce projet dispose aujourd'hui de plusieurs codes de simulation, dont un code système permettant d'évaluer le comportement d'un tel réacteur dans le cadre d'hypothèses simplificatrices.

Nommé « LiCore » pour *Liquid Core*, il présente le grand avantage d'être particulièrement rapide, mais il nécessite que l'intégralité du scénario de simulation soit défini à l'avance. Durée de simulation, insertion de réactivité, suivi de charge, débit des pompes… tous les paramètres et leurs évolutions doivent être saisis avant de lancer le calcul.

L'objectif de ce stage est d'adapter LiCore et de développer les éléments nécessaires à son utilisation sur la plateforme ALICES®, les fonctionnalités de ce logiciel permettant un pilotage en temps réel. Après avoir présenté les fonctionnements du MSFR, du code LiCore et d'ALICES®, ce rapport expose les différentes étapes du développement de ce projet.

I. Présentations

1) Le MSFR

a) Concept du réacteur

Les réacteurs actuellement en fonctionnement sont alimentés par du combustible solide, usiné sous forme de pastilles regroupées en crayons combustible. Ce n'est pas le cas du MSFR illustré Figure 1, qui utilise environ 18m^3 de combustible *liquide*, jouant également le rôle de caloporteur. Une autre différence notable avec les réacteurs actuels est le spectre énergétique des neutrons. En effet, ce concept n'emploie pas de modérateur, d'où un spectre qualifié de *rapide*. Les neutrons ne sont pas modérés comme c'est le cas dans les réacteurs à eau. Ce type de spectre présente plusieurs avantages qui seront présentés dans la prochaine section.

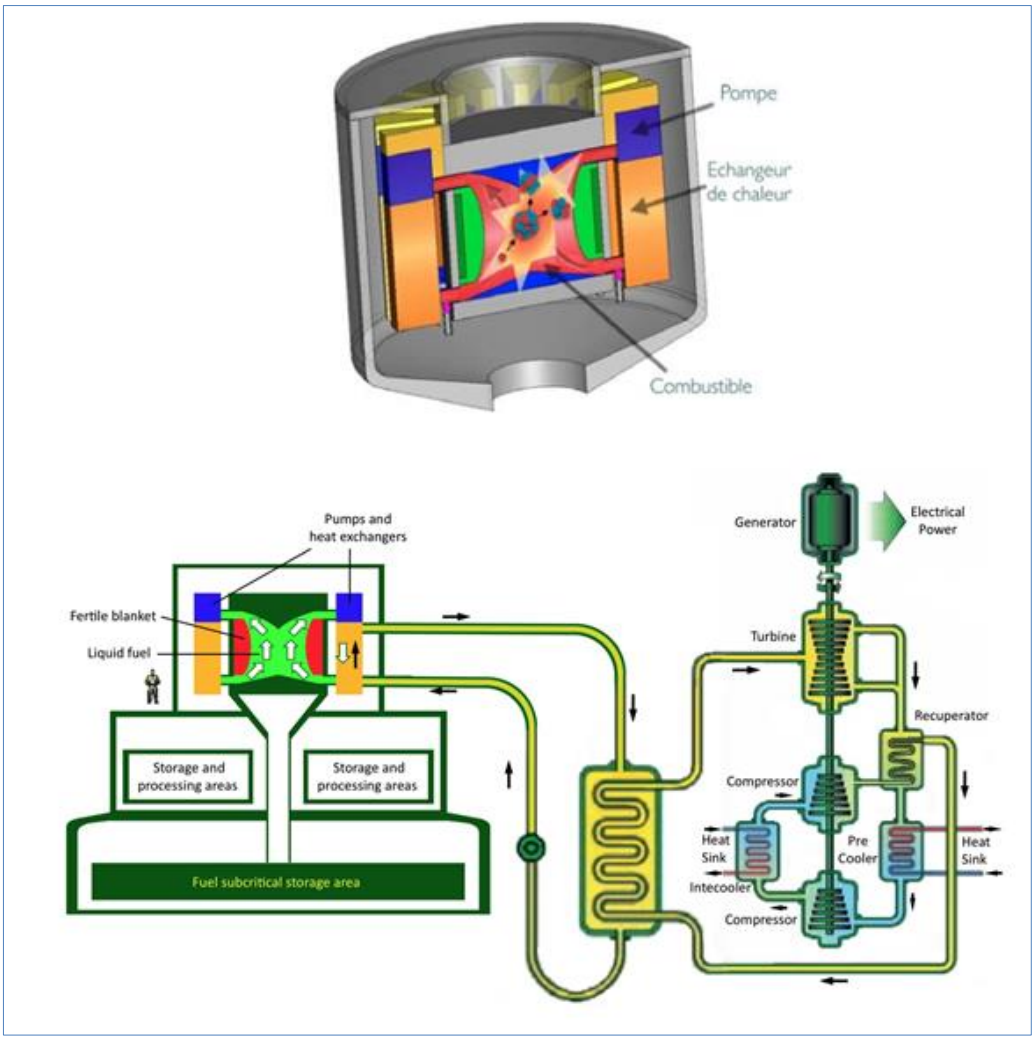


Figure 1 : Représentation schématique du MSFR [3] [4]

Les espèces radioactives alimentant la réaction en chaîne sont en solution dans un sel en fusion mis en mouvement par des pompes, une pour chacune des 16 boucles de circulation. Le mélange sert donc à la fois de combustible dans un cœur de 9m^3, et de caloporteur lorsqu'il passe dans les échangeurs de chaleur situés autour.

Dans ces échangeurs, l'énergie est transmise à un circuit intermédiaire, puis au circuit chargé

de la conversion en énergie électrique. Le sel utilisé est principalement composé de Fluor et Lithium car il est à la fois chimiquement inerte et stable sous irradiation. En outre il possède de bonnes caractéristiques hydrauliques. Un sel de ce type permet de fonctionner à pression ambiante à des températures aux alentours de 1000K.

Ce réacteur est alimenté par le cycle Thorium-Uranium, c'est-à-dire que le principal noyau fissile présent est ^{233}U obtenu à partir de ^{232}Th après absorption d'un neutron :

$$\mathrm{n} + {}^{232}_{90}\mathrm{Th} \rightarrow {}^{233}_{90}\mathrm{Th} \xrightarrow[22\ min]{\beta^-} {}^{233}_{91}\mathrm{Pa} \xrightarrow[27\ j]{\beta^-} {}^{233}_{92}\mathrm{U}$$

Ce cycle de fonctionnement implique d'avoir une espèce fissile présente dans le cœur afin d'initier la réaction en chaîne. Il peut s'agir de ^{233}U produit par ailleurs, ou encore de ^{235}U comme dans les réacteurs à eau. Une fois en fonctionnement, l'utilisation d'une couverture fertile au sein du cœur permet la *surgénération* (la capacité d'un système à produire davantage de noyaux fissiles qu'il n'en consomme).

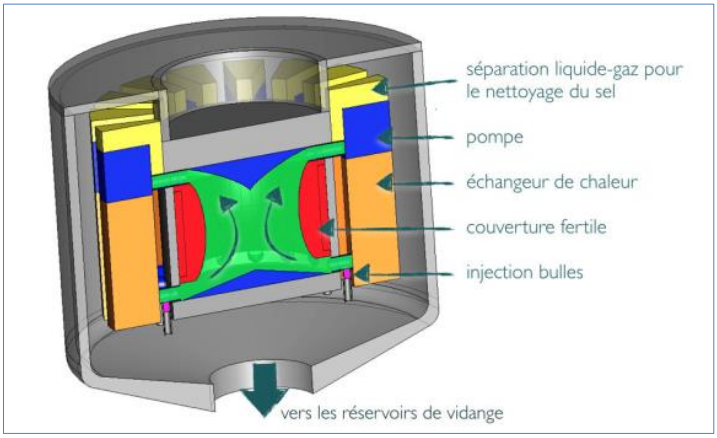


Figure 2 : Représentation schématique du circuit du combustible dans le MSFR

Le fonctionnement engendre la production de produits de fissions. Ceux qui ne sont pas solubles sont évacués du cœur à l'aide d'un bullage de gaz qui entraînent les déchets en dehors du combustible grâce à un séparateur gaz/liquide présent dans le circuit. Par ailleurs, une petite partie du sel (quelques dizaines de litres par jour) doit être retraitée afin d'en extraire les lanthanides, d'en ajuster la teneur en noyaux fissiles et les propriétés redox.

b) Avantages

D'une manière générale ce système parait simplifié par rapport à un réacteur à combustible solide : on élimine à la fois la complexité de l'usinage des pastilles et crayons combustible, des plans de chargements, du maintien à haute pression pour fonctionner (150 bars pour un réacteur à eau pressurisée).

Mais un des avantages principaux que présente ce concept est sa sureté intrinsèque, offerte notamment par l'état liquide du combustible. En effet, en plus de l'effet Doppler (contre réaction thermique également présent dans les autres types de réacteur) les effets de dilatation permettent des coefficients de contre réaction très négatif. La dilatation réduit la densité du milieu, donc les sections efficaces macroscopiques, et une partie du combustible se retrouve en dehors du cœur via un trop-plein.

L'état liquide permet également de supporter une plus large plage de température que les crayons combustible. Et en cas de problème majeur, un simple écoulement vers des réservoirs de

vidange permet de l'étaler sur une grande surface afin de stopper la réaction en chaîne au sein du mélange et d'évacuer la chaleur de façon totalement passive.

Dans un réacteur à eau l'utilisation de l'ensemble des crayons n'est pas homogène, imposant de prévoir des plans de chargement pour utiliser le combustible au mieux. Ici cette contrainte n'existe pas puisque l'ensemble du combustible est homogénéisé par brassage.

Par ailleurs on bénéficie d'un fonctionnement en spectre rapide, pour lequel les sections efficaces sont très réduites, ce qui permet de diminuer drastiquement l'empoisonnement du combustible par des produits de fissions : un réacteur à spectre thermique nécessiterait non pas 10 à 40 litres mais plusieurs mètre cube de combustible retraité par jour. Des neutrons aussi énergétiques peuvent endommager les équipements, mais cet effet est réduit par les diffusions inélastiques du fluor.

Enfin, il s'agit d'un système compact : un cœur de 9 m^3 et 18 m^3 de combustible pour environ 2m de haut, bien plus facile à usiner qu'une cuve (4m) de réacteur à eau devant fonctionner à haute pression. On peut alors envisager des applications de type « Small Modular Reactor » : des cœurs de taille réduite qui pourraient permettre une production énergétique locale.

2) Code système LiCore

NB : Dans la thèse d'Axel Laureau [5], les équations et hypothèses du code LiCore sont présentées sous le nom du modèle neutronique développé : « Cinétique Point par Zone » (voir aussi [6])

a) Hypothèses de simulation

- Rappels de neutronique

Dans un système nucléaire, on appelle *criticité* le maintien de la réaction en chaîne sans emballement. Dans cet état, le flux de neutrons au sein du réacteur est constant au cours du temps. L'établissement et le maintien de la criticité dépendent de paramètres dits « matériaux » d'une part, et géométriques d'autre part.

La composition détermine les sections efficaces de réaction, donc à la fois la consommation et la production de neutrons. Mais les fissions ne sont pas les seules sources. Par la suite certains produits de fissions émettent eux aussi des neutrons lorsqu'ils décroissent. Ces *neutrons retardés* (par opposition aux neutrons *prompts*) sont donc produit de quelques secondes à quelques minutes après la fission initiale. C'est cette fraction de neutrons retardés qui permet le pilotage des réacteurs : leur temps caractéristique d'évolution permet aux contre réactions de réagir en cas de sous ou surcriticité afin de revenir à une situation stable.

D'autre part, la forme du système est elle aussi capitale puisqu'elle a un impact direct sur la quantité de neutrons ayant la possibilité de sortir du milieu actif sans avoir réagi. Ainsi, un neutron émis proche du bord du système aura une plus faible probabilité d'engendrer une fission qu'un autre qui aurait été émis au centre. Cette différence est quantifiée par ce que l'on appelle l'*importance*, et il est possible de calculer une carte de l'importance : le *flux adjoint*. On peut alors pondérer la fraction de neutrons retardés β et le temps entre deux générations de neutrons Λ, afin d'obtenir leurs valeurs effectives β_{eff} et Λ_{eff}

- Retour sur le MSFR

Dans le cas qui nous intéresse ici, nous sommes confrontés à une difficulté supplémentaire : la circulation du combustible. Dans un combustible solide, les produits de fissions sont très peu

mobiles, donc on peut supposer que les neutrons retardés sont émis au même endroit que les neutrons prompts. Mais ici, le temps de circulation (3 à 4 s environ), est du même ordre de grandeur que la période de décroissance de certains des précurseurs de ces neutrons retardés.

Pour effectuer ce calcul dans le code LiCore on se place dans l'approximation de cinétique point, qui découple les variables spatiales et temporelle. Afin de gagner en efficacité on calcule dans un premier temps la distribution des précurseurs, régis par le transport, à laquelle on applique ensuite la correction de l'importance. Ainsi les équations de la cinétique point donnent :

L'évolution de la densité neutronique au cours du temps

$$\frac{dn(t)}{dt} = \frac{\rho(t) - \beta}{\Lambda_{eff}} n(t) + \sum_f w_f(\boldsymbol{r}, t) \lambda_f P_f(\boldsymbol{r}, t) d\boldsymbol{r}$$

L'évolution de chaque famille de précurseur f :

$$\frac{dP_f(\boldsymbol{r}, t)}{dt} = \frac{\beta_f}{\Lambda_{eff}} n(t) \Phi(\boldsymbol{r}) - \lambda_f P_f(\boldsymbol{r}, t)$$

$\Phi : flux\ de\ neutrons$
$\rho : réactivité$
$\beta : fraction\ de\ neutrons\ retardés$
$\Lambda_{eff} : temps\ de\ génération\ effectif$
$\lambda_f : constante\ de\ décroissance\ des\ précurseurs\ f$
$w_f : rapport\ d'efficacité\ entre\ un\ neutron\ retardé\ f\ et\ un\ neutron\ prompt$
$\beta_f : fraction\ de\ neutrons\ retardés\ de\ la\ famille\ f$

Comme on peut le voir, ces deux équations sont fortement couplées. Etant à la recherche d'ordres de grandeur caractéristiques, et pas d'un couplage précis, on introduit des approximations permettant de de rendre le calcul plus rapide en simplifiant.

On suppose tout d'abord que la forme du flux dans la zone active est invariante durant toute la simulation, proportionnelle à une fonction cosinus axialement et à la fonction de Bessel J_0 représentée en Figure 3.

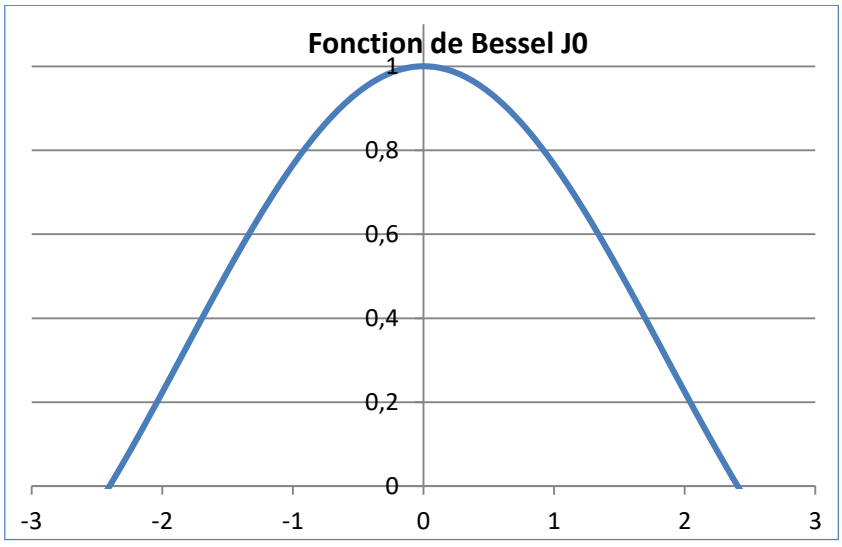


Figure 3 : Représentation de la fonction de Bessel

On suppose également que les contre réactions Doppler et densité sont constantes selon la température, donnant un coefficient de contre réaction total $\frac{dk}{dT} = -8pcm/K$ k étant le coefficient de multiplication de la population neutronique. Cette contre-réaction est utilisée localement, cellule par cellule lors du calcul neutronique, et non pas globalement.

- Hydraulique

Afin là aussi de rendre les calculs plus rapides, le cœur et le circuit sont modélisés par un ensemble de sections dans lesquelles le combustible suit un mouvement piston : il circule de manière unidirectionnelle et monophasique.

Comme illustré sur la figure 4, il y plusieurs types de section. L'une d'entre elles, située en haut de l'échangeur, impose le débit tout en compensant la dilatation thermique via un trop-plein. Une fois dans l'échangeur, le refroidissement peut être fait selon deux modes : à température du fluide intermédiaire fixée ou bien à puissance extraite fixée.

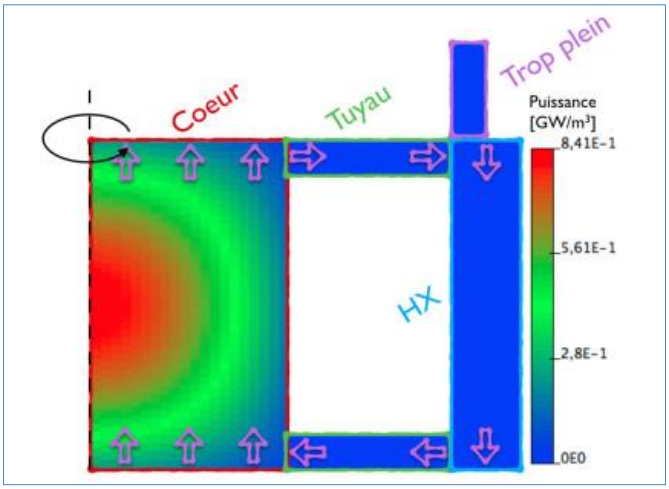


Figure 4 : Découpage du circuit combustible du MSFR en différentes sections dans le modèle CPZ/LiCore ([5], chapitre 5, figure 5.1)

A la sortie de l'échangeur, le combustible passe dans une partie tuyau, avant d'arriver dans le cœur en lui-même. Ce dernier est non seulement discrétisé axialement comme les autres types de section, mais aussi radialement afin de rendre compte du champ de température. De plus, le champ de vitesse est imposé de façon à avoir une température homogène au sommet de la zone, au moment de reboucler sur le tuyau du haut puis l'échangeur.

En outre, LiCore comprend un vase d'expansion au-dessus et un circuit de vidange en bas (non représentés ici). Le combustible s'écoule par ce dernier s'il dépasse une certaine température, fixée par la température de fusion d'une bonde placée en bas de l'échangeur.

b) Principales classes de simulation

- Notions sur la programmation orientée objets

Le code LiCore a été développé dans le langage Java, son fonctionnement repose donc sur des *objets*. En programmation, un objet est défini par sa *classe* qui comporte un certain nombre d'attributs et de méthodes. On dit que l'objet est une instance de la classe.

Les attributs sont les caractéristiques de l'objet, ils forment en quelque sorte sa carte d'identité. Ses méthodes quant à elles permettent de traiter les données qui lui sont fournies

(comme dans un programme 'classique') et d'interagir avec d'autres objets.

Cette façon de programmer permet d'encapsuler et de protéger le code et les données. Elle offre une grande souplesse puisque l'on peut mettre à jour une classe sans avoir à modifier l'intégralité du code. En effet, les objets extérieurs ne voient pas le fonctionnement interne de la classe que l'on modifie, donc tant que les interactions ne changent pas (noms, paramètres et type de retour des méthodes) ils ne sont pas affectés.

Une autre notion importante est celle de l'hérédité : une classe peut hériter des attributs et méthodes d'une autre, on la désigne alors comme classe 'fille'. Ce principe permet d'aller du plus haut niveau de généralisation possible aux cas plus particuliers, en quelque sorte à la manière d'un arbre phylogénétique. Par exemple : une classe Félix peut être une instance d'une classe ChatDeGoutiere héritée d'une classe Chat, elle-même fille de la classe Félin qui hérite de la classe Mammifère. Et cette classe Mammifère serait également mère de Canin et Bovin. Il est à noter qu'en Java une classe peut avoir autant de filles que l'on veut, mais qu'elle ne peut hériter que d'une seule et unique mère.

- Structure de LiCore

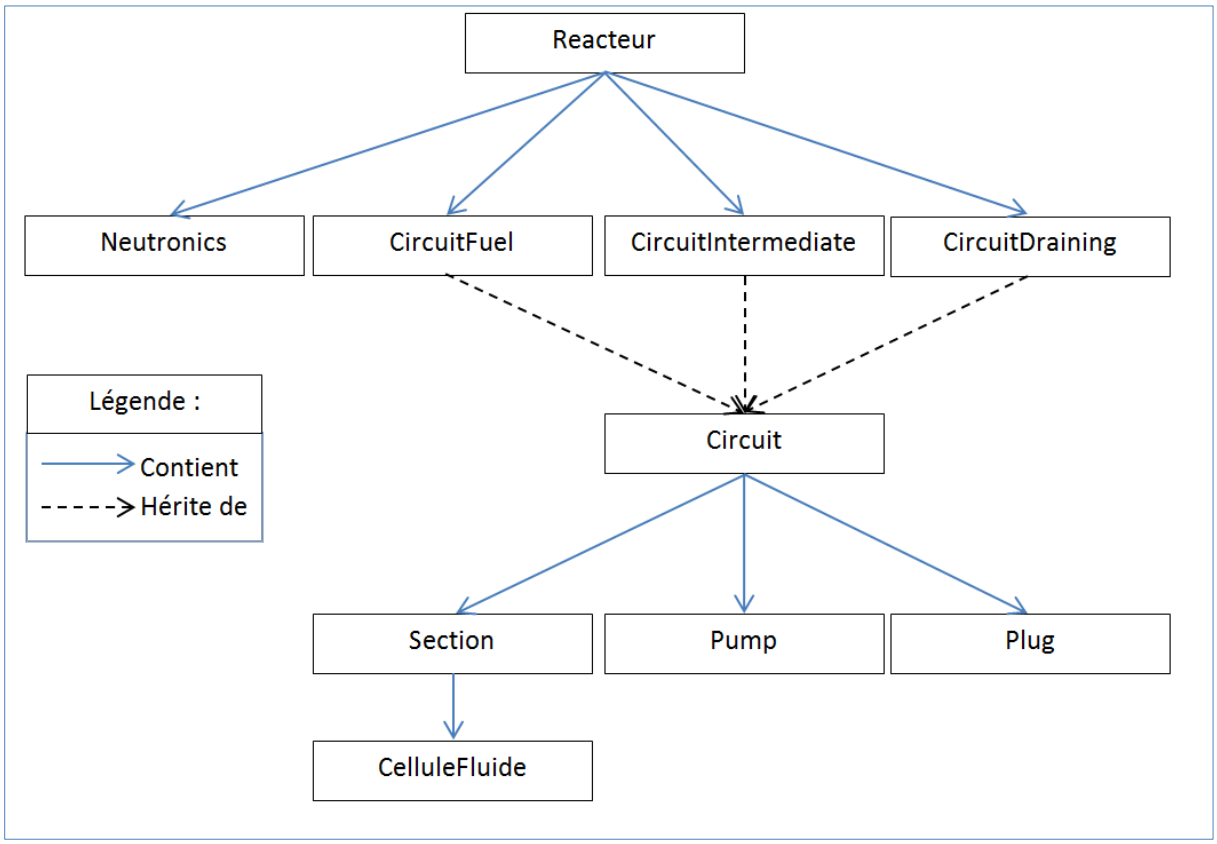


Figure 5 : Schéma de structure des principales classes de LiCore

Comme on le voit sur la Figure 5, la programmation orientée objet permet une structure relativement intuitive qui permet de refléter la réalité : un véritable réacteur contient effectivement trois parties de circuit composées de sections dans lesquelles circule du fluide, et l'on retrouve cette structure dans LiCore.

De plus, bien qu'ayant des rôles et comportements distincts, les différents types de circuit

ont de nombreux points communs. Leurs comportements ont donc été généralisés et hérités d'une classe mère Circuit.

A ces classes dédiées aux écoulements il faut ajouter le second aspect de la modélisation souhaitée : la neutronique. La classe Neutronics contient tous les paramètres et méthodes permettant de calculer la population neutronique et d'en déduire les différentes grandeurs qui nous intéressent.

Afin de les présenter plus en détail, les classes sont ici mises sous la forme de tableaux comme celui-ci-contre. Dans cette section, seuls les éléments nécessaires à la compréhension de la modélisation sont indiqués. La représentation graphique du réacteur est quant à elle détaillée en section suivante.

Nom de la Classe
<Type d'attribut1> <nom d'attribut1> **…**
<Type de retour1> <nom de méthode1> **…**

Réacteur	
String	**nom**
double	**temp_initial**
double	**temps**
double	**temps_max**
double	**puiss_extraite**
Neutronics	**module_neutronique**
CircuitFuel	**circuitCombustible**
CircuitIntermediate	**circuitIntermediaire**
CircuitDraining	**circuitVidange**
TimeStepManager	**time_step_manager**
void	**creerGeometrie**
AffParam	**avance_de_dt**

Tableau 1 : Classe Java Reacteur

Comme on l'a vu, la classe Reacteur comprend des instanciations des classes permettant la modélisation, mais également différents paramètres numériques. Le code LiCore simule une durée déterminée à l'initialisation via *temps_max* et *temps_initial.* On attribue à cette dernière une valeur négative, correspondant à une durée de mise à l'équilibre du réacteur en fonction d'une température de consigne. Les différents événements du scenario (suivi de charge, insertion de réactivité…) sont programmés ensuite, dans les temps positifs.

La méthode *creerGeometrie* permet d'initialiser le réacteur à partir d'un fichier input. Ce fichier renseigne tous les paramètres nécessaires à la simulation, notamment les dimensions et la discrétisation du réacteur. La méthode *avance_de_dt* quant à elle permet de calculer l'évolution sur un pas de temps, et retourne l'ensemble des grandeurs que l'on souhaite afficher.

Enfin, le Reacteur possède également une instance d'une classe TimeStepManager. Comme son nom l'indique, son rôle est de gérer le pas de temps adaptatif de la simulation.

Neutronics	
double	beta0
double	dk_dT
double	population
double	populationOld
double	generationTime
double	core_radius
double	getRho
double	realise_neutronique

Tableau 2 : Classe Java Neutronics

Sur le même principe, la classe Neutronics récupère les paramètres nécessaires à la simulation à partir de l'input, tels que la fraction de neutrons retardés, la contre-réaction thermique et le temps de génération par exemple.

La méthode *realise_neutronique* est la première appelée par *avance_de_dt* de Reacteur. Elle calcule la réactivité via la méthode *getRho* en fonction de la température. Les formes du flux et de la répartition des précurseurs étant considérées fixes, on peut en déduire la population neutronique, la puissance thermique générée, et le champ de température au sein du réacteur.

Circuit		CircuitFuel	
Pump	pump	Section	hx
Plug	plug	Section	pipe_haut
Double	puiss_hx	Section	pipe_bas
Section	super_hx;	Section	trop_plein
Section	super_other_hx;	ArrayList<Section>	sections_coeur
Section	super_trop_plein;		
Section	super_fin_de_section;		
Section	addSectionDroite	ArrayList<Section>	creerSectionsEnCoeur
Section	addSectionCyl		
Void	avance_de_dt		
CircuitIntermediate		**CircuitDraining**	
Section	hx_2	Section	tuyau
Section	hx_2_haut	Section	cuve
Section	hx_2_bas		
Section	hx_2_hx		
Section	hx_2_tp_plein		

Tableau 3 : Classes Java Circuit et ses filles

La classe Circuit a deux principaux rôles. D'une part elle définit de manière globale les méthodes permettant d'initialiser le Reacteur par instanciations successives d'objets Section via *addSectionDroite* ou *addSectionCyl* selon l'emplacement (en cœur ou en tuyau). Ce sont ces

méthodes qui sont utilisées par *creerGeometrie* de la classe Reacteur, par l'intermédiaire du constructeur du Circuit en question. D'autre part, elle définit *avance_de_dt*, qui permet de calculer les différents échanges de chaleur (entre circuits et entre sections) et la circulation des objets CelluleFluide.

Chaque type de circuit a des attributs supplémentaires par rapport à Circuit correspondant à ses particularités. Le circuit de vidange est le plus simple, ne contenant qu'un tuyau faisant le lien entre le circuit combustible et la cuve de refroidissement. Le circuit intermédiaire est composé de 3 tuyaux (*hx_2* ; *hx_2_haut ; hx_2_bas*), d'un échangeur *hx_2_hx* et d'un trop plein *hx_2_tp_plein* permettant la dilation du sel.

Enfin, le circuit combustible est composé de deux tuyaux *pipe_haut* et *pipe_bas* qui relient l'échangeur *hx* et le cœur. La construction de ce dernier dépend de la discrétisation choisie, c'est pourquoi le nombre de sections en cœur n'est pas fixé, et que les Section sont listées dans une *ArrayList<Section>*[1].

CelluleFluide	
double	**temperature**
double	**beta_dilatation**
double	**T_ref**
double	**masse**
double	**densite_ref**
double	**Cp**
double	**fraction_flux**
double[]	**tab_prec**
double[]	**tab_beta**
double[]	**tab_lambda**
void	**set_time_step_flux_and_add_weighed_values**
void	**apply_this_power_and_precursor_production**
double	**refroidit**
double	**echange_avec**
void	**ingurgite**

Tableau 4 : Classe Java CelluleFluide

Les propriétés du sel (température, teneur en précurseurs, densité…) sont portées par des instanciations de la classe *CelluleFluide* (voir tableau 4). L'emplacement et la forme fixée du flux permettent de connaître l'attribut *fraction_flux* et d'en déduire la puissance thermique, la température et les productions et consommations de précurseurs. Ces derniers sont classés par famille, chacune d'entre elles ayant un temps de décroissance et une fraction de neutrons retardés caractéristiques, listés dans les tableaux *tab_beta* et *tab_lambda*.

Cette classe contient plusieurs méthodes de calcul, permettant de calculer les différents échanges d'énergie et matière, ainsi que les productions et consommation de précurseurs.

[1] *Une ArrayList<E> désigne une liste dynamique d'éléments E, c'est-à-dire notamment que sa taille peut être modifiée, contrairement à un tableau classique. Le E, peut désigner n'importe quelle classe Java, et dans notre cas il s'agit de Section.*

LeGrandGestionnaire	
ArrayList<Reacteur>	reacteur
JFrame	fenetre
JPanel	fond
PanDessinReact	dessin
double	temps_max
void	do_your_job
void	aff_react
void	clean_curves

Tableau 5 : Classe Java LeGrandGestionnaire

Par ailleurs, on souhaite dans le cadre de LiCore être capable d'effectuer des études paramétriques, donc pouvoir simuler plusieurs cœurs à la fois. C'est pourquoi la simulation est pilotée par une classe supplémentaire : *LeGrandGestionnaire* (cf. tableau 5).

Cette classe possède en particulier une *ArrayList<Reacteur>*, les méthodes de stockage et d'affichage des variables, et *do_your_job* qui contient la boucle de simulation, ainsi que les objets permettant l'affichage du cœur. Enfin, une classe *Bourreau* sert à exécuter la simulation : c'est elle qui contient la méthode main.

c) *Visualisation du MSFR*

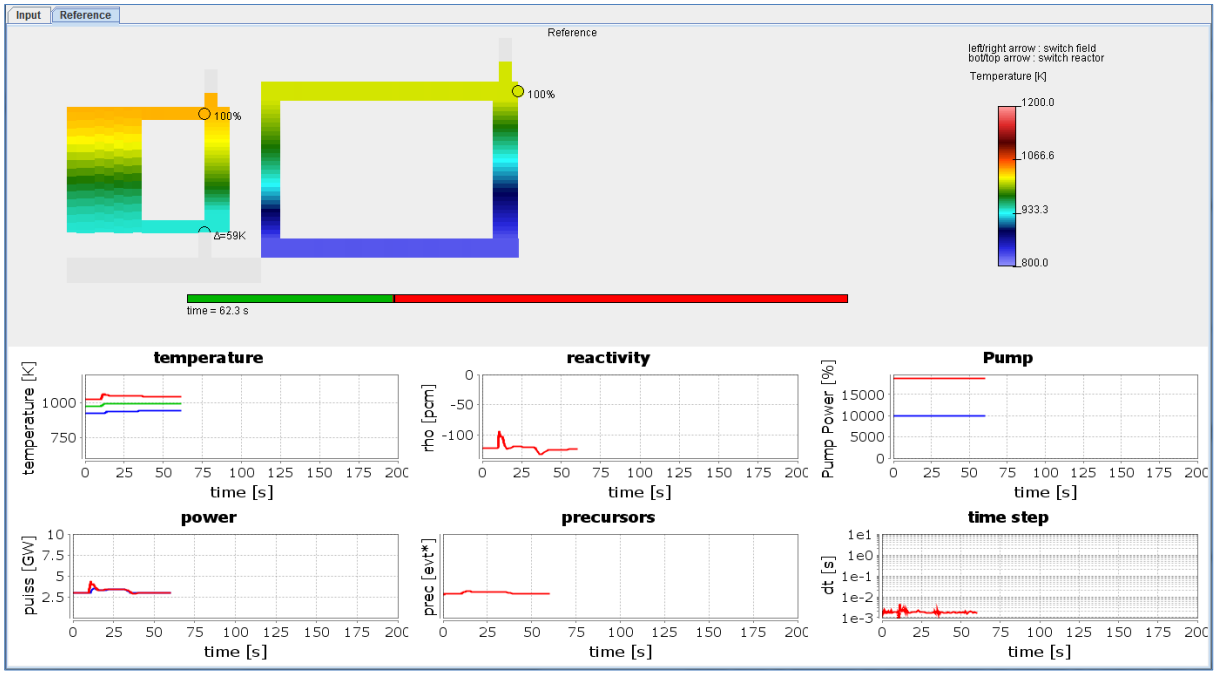


Figure 12 : Représentation du MSFR en cours de simulation

En Java, la création d'une fenêtre suit un principe de poupées russes : l'objet *JFrame* correspondant à la fenêtre en elle-même est le contenant principal. Il contient un ou des panneaux *JPanel*, dans lesquelles peuvent être ajoutés des widgets (boutons, champs de saisi) des dessins ; ou encore d'autres panneaux.

Dans le code LiCore, la représentation graphique du réacteur, illustrée Figure 12, repose principalement sur la classe *PanDessinReact*, qui hérite de la classe *JPanel*. Le tracé des courbes est géré par un autre panneau, mais la gestion des courbes étant par la suite assurée par Alices®, son fonctionnement ne sera pas détaillé ici.

PanDessinReact	
ArrayList<ArrayList<ArrayList<ArrayList<Double>>>>	liste_rea2mod2t2val
ArrayList<Double>	liste_temps
ArrayList<Double>	liste_rectangle
double[]	xy_lim
double	time
int	Tmin ; Tmax
void	affiche_react
void	paintComponent
void	do_legend
void	do_timeline

Tableau 6 : Classe Java PanDessinReact

Le type de l'attribut *liste_rea2mod2t2val* peut sembler complexe, il s'agit en fait de la liste des champs scalaires affichables des réacteurs simulés (température, puissance,…). On accède aux données à partir d'une succession d'index : *rea2mod2t2val [reacteur] [mode] [time] [id]* avec :

- *reacteur* : numéro du réacteur à afficher (plusieurs peuvent être simulés simultanément)
- *mode* : choix du type de champ à afficher - température, puissance, décroissance de précurseurs
- *time* : choix du temps associé à l'affichage, par défaut il s'agit du dernier calculé mais l'utilisateur peut choisir de visualiser un état précédent
- *id* : liste des éléments à afficher (coordonnées des points, valeurs, etc…)

LiCore offre la possibilité de visualiser l'état du réacteur à n'importe quel moment du scénario lors de la simulation et une fois qu'elle est terminée, ce qui passe par l'enregistrement de chaque instant dans le terme *liste_temps* et le fait d'avoir un terme *time* itéré à chaque pas. Les coordonnées des limites du dessin sont données par *xy_lim*, et *Tmin* et *Tmax* permettent de construire la légende.

Chaque élément du dessin (réacteur, ligne de temps et légende) est créé par une fonction qui lui est propre, puis le tout est redessiné via la méthode *paintComponent* hérité de *JPanel*

La section suivante présente la plateforme de programmation Alices sur laquelle le code LiCore a été intégré lors de ce stage.

3) ALICES®

Le principe de la conception de simulateurs sous Alices® est semblable à la programmation orientée objets. Il y a trois niveaux imbriqués, chacun ayant sa façon de communique. Là encore, chaque fonctionnalité est encapsulée : vu de l'extérieur, un module est une boite noire disposant d'entrée(s) et sortie(s). Les caractéristiques de chacun de ces niveaux sont décrites dans la documentation Corys [5].

NB : Alices® permet deux types de résolution : séquentielle et différentielle. Cette dernière n'ayant pas été utilisée au cours de ce stage, elle ne sera pas exposée dans ce rapport.

a) Niveau 1

L'objet de Niveau 1 représente la brique élémentaire des simulateurs fonctionnant sous Alices®. Ils effectuent les calculs à proprement parler. Il peut s'agir d'opérations (logiques ou analogiques) plus ou moins complexe effectuée à chaque pas de temps.

Un objet Nv1 est défini par son code de calcul, une fiche de variables et un dessin, dont on a un exemple en Figure 6 :

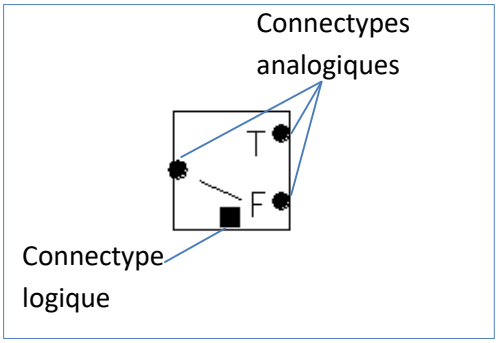


Figure 6 : Dessin d'un sélecteur deux branches

Les *connectypes* sont le moyen de communication des objets Nv1. En les connectant les uns avec les autres ils permettent l'échange de données tout en garantissant la cohérence des types (logique ou analogique) entre entrées et sorties (cf. Figure 7).

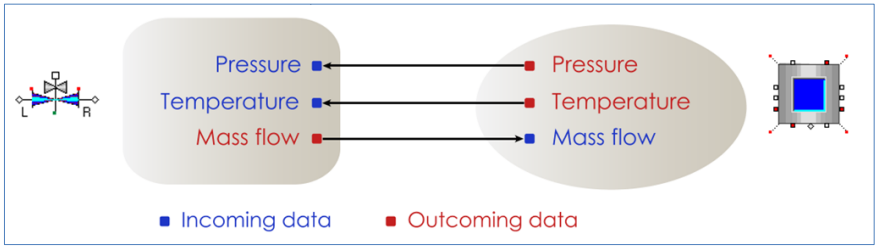


Figure 7 : Exemple de communication entre une vanne et un réservoir

Lors de la conception de la fiche de variables, une distinction importante est faite entre *paramètres* et *variables.* Ces dernières sont calculées à chaque pas de temps, tandis que les paramètres sont fixés à la conception et restent constants. Les modules de Niveau 1 comportent également les *prototypes* qui sont à la base de la conception des IHM des simulateurs. Tous ces différents objets sont triés en bibliothèques, et on peut alors les utiliser pour construire les objets de Niveau 2.

b) Niveau 2

Les objets de Niveau 1 sont assemblés et connectés entre eux au sein de *documents* pour les calculs, et d'*images* pour les IHM. La programmation se fait ici de façon graphique : les objets sont ajoutés depuis les bibliothèques en cliqué-déposé, puis des lignes de connections sont tracées entre les connectypes des différents objets, comme illustré en Figure 8.

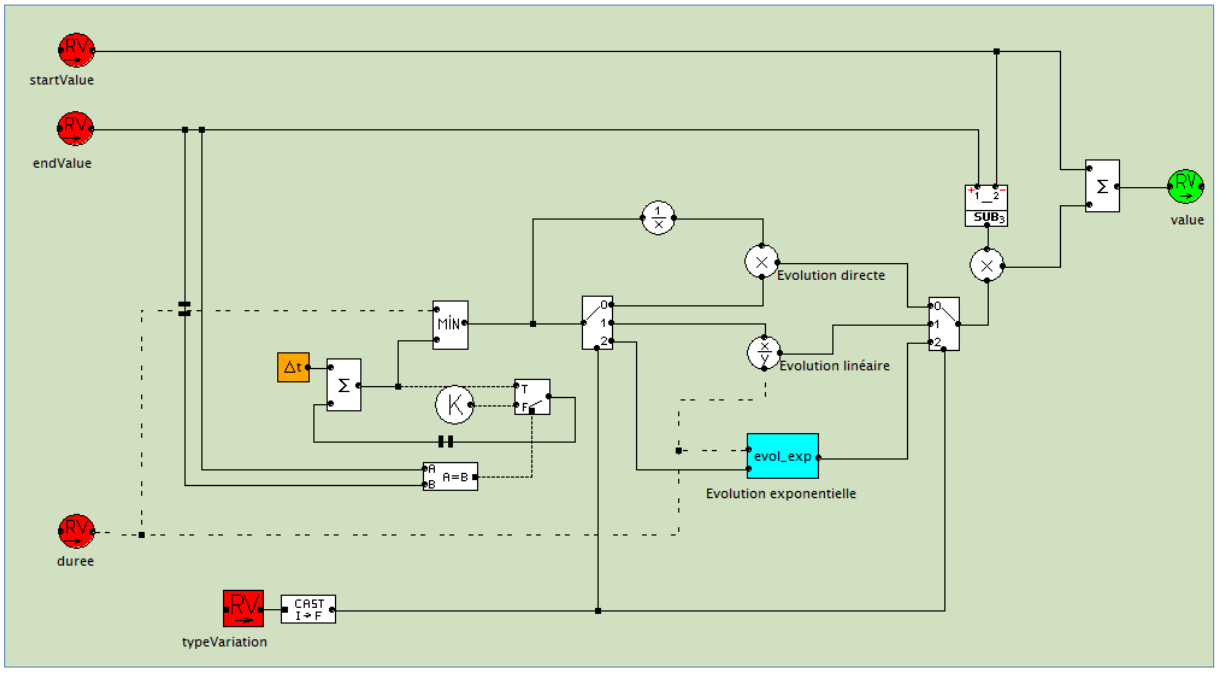
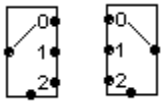


Figure 8: Exemple de document Niveau 2

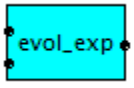
Sans chercher à détailler le fonctionnement de ce document que nous reverrons en partie III, on voit sur la Figure 8 un document contenant différents types d'objets Niveau 1 :

> Fonctions mathématiques : elles ont chacune une ou plusieurs connections d'entrée analogique pour une sortie analogique :

> Objets de la bibliothèque « Select » permettant de faire des disjonctions de cas : une connections d'entrée permet de faire un choix entre différentes entrées ou sorties analogiques.



> Macro objet : ces objets encapsulent une fonction nécessitant plusieurs objets, ce qui permet notamment d'améliorer la lisibilité du document



> Import/export de variables d'échange, réelles ou entières

Les variables d'échange, définies dans la fiche Nv2 des objets, constituent le moyen de communication entre les différents modules de Niveau 2. Au sein d'Alices®, le nom des variables suit un format de la forme « jeton1:jeton2:jeton3 », chaque jeton étant une chaîne de caractères. Cette écriture permet aux différents modules de communiquer correctement une fois configurés, comme nous le verrons dans la section suivante.

Les variables d'échanges sont séparées en trois catégories : variables d'*import*, d'*export* et d'*état*. Les deux premières catégories sont explicites, mais il convient de s'attarder sur la notion de variable d'état, très importante dans Alices®.

Les variables d'états sont un sous-ensemble de variables permettant de totalement définir l'état d'un système. Ainsi en les sauvegardant dans un fichier nommé *cliché*, on dispose d'un état initial du système qui est parfaitement reproductible. Cela permet également de sauvegarder des états pertinents pour la formation d'étudiants ou d'opérateurs, etc… Un simulateur complet pouvant manipuler un très grand nombre de variables, il est important de déterminer le groupe de variables qui soit juste nécessaire et suffisant.

Par ailleurs on remarque des boucles dans le document de la Figure 7, l'une d'entre elles est spécifiquement illustrée en Figure 9.

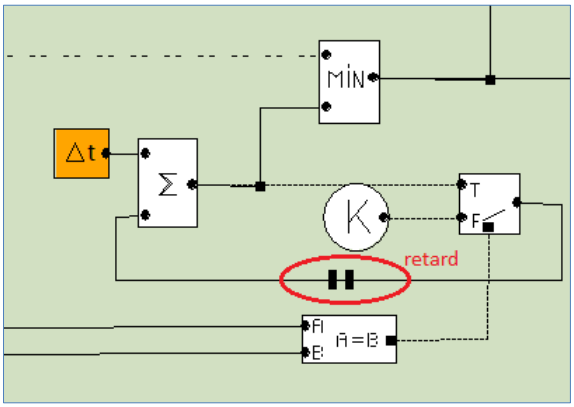


Figure 9 : Boucle de retard

Dans cette situation, on cherche notamment à incrémenter le temps à chaque pas, on a alors une entrée du module d'addition qui dépend de la sortie de ce même module. Ce problème est résolu grâce à un *retard* appliqué sur la boucle de retour. Il suffit alors d'initialiser la valeur de l'entrée concernée pour le premier pas.

c) Niveau 3

- Configuration de simulation

Dernière étape de la conception du simulateur : les modules sont instanciés à partir de ces documents Nv2 et/ou à partir des modules Nv3 (exposés dans la sous-section suivante) et regroupés en une *configuration*. Elle peut alors être lancée depuis le moniteur Alices®.

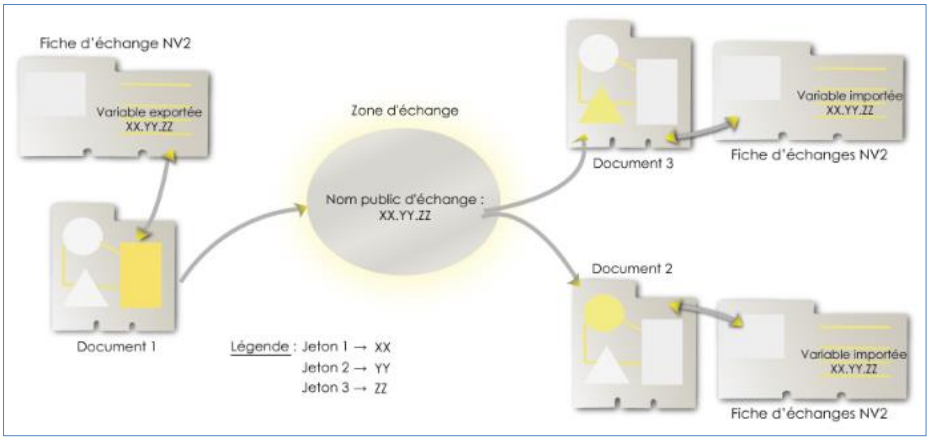


Figure 10 : Schéma de déclaration et d'échange d'une variable ([5], figure 2.22)

Les modules communiquent leurs variables via la *zone d'échange*. A chaque fin de pas de temps, les variables exportées sont envoyées dans la zone d'échange, puis récupérées par les modules qui les importent au début du pas suivant. Seules exceptions : les variables dites *asynchrones* ne sont exportées que lorsque leur valeur est modifiée. De plus, chaque variable ne peut être exportée qu'une unique fois, mais importée par autant de modules que souhaité.

La Figure 11 suivante illustre la configuration finale de ce projet. Les modules représentés par des ovales verts sont répartis entre deux processus de calculs parallèles appelés « Séquences ». L'une est consacrée à la simulation, l'autre au pilotage. A l'exception du module « GestionSimu » qui est natif dans Alices®, ils ont tous été développés au cours de ce projet. Leurs rôles et fonctionnement seront détaillés dans la suite de ce rapport.

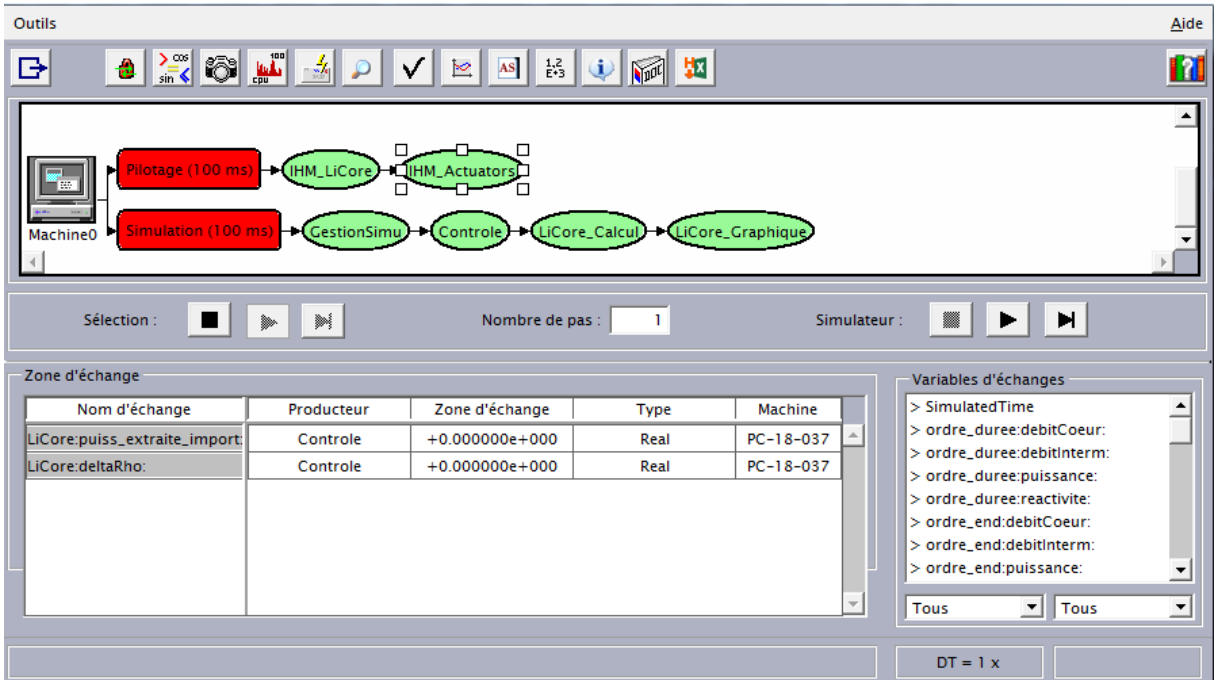


Figure 11 : Configuration Niveau 3 de LiCore

Ici le moniteur Alices® prêt à lancer une simulation LiCore. La partie IHM de pilotage est traitée dans une séquence -donc un processus- à part. Le moniteur permet de visualiser des variables de notre choix dans la zone d'échange, et de voir l'ensemble des variables échangées par un module donné dans le volet en bas à droite.

Enfin, les différents boutons en haut du moniteur offrent de nombreuses fonctionnalités. Certains servent à la gestion des courbes et des clichés, ou encore à contrôler si le temps de calcul reste inférieur au temps réel. D'autre sont principalement utiles lors du développement du simulateur, permettant par exemple de forcer les valeurs de certaines variables.

- Structure d'un module Nv3

Il est possible de définir un module sans passer par les étapes d'objets Nv1 et documents Nv2, on parle alors de module Nv3. Ce type de module est codé en C++, un langage orienté objet autre que Java que l'on a vu avec LiCore, et contient obligatoirement une classe hérité de *Nv3BaseDeModule*, comprise dans Alices® (voir Tableau 7).

Module	
ofstream	**mLogFile**
<variables d'import>	
<variables d'export>	
<variables d'état>	
Nv3BaseDeModule*	**Nouveau**
void	**Initialisation**
void	**Traitement**
void	**Terminaison**
void	**EnregistrerCliche**
void	**ChargerCliche**

Tableau 7 : Classe principale d'un module Nv3

Les noms de ces méthodes sont explicites : *Initialisation* est appelée avant de commencer la simulation pour initialiser les variables qui en ont besoin, *Traitement* effectue un pas de temps, et *Terminaison* arrête le processus de simulation. Il faut de plus définir toutes les variables d'échanges du module. L'attribut *mLogFile* quant à lui désigne un ''fichier log'', c'est-à-dire un fichier dans lequel sont récupérées les informations utiles pendant le développement du module (messages d'erreur etc…). D'une manière générale il n'est pas obligatoire de redéfinir *EnregistrerCliche* et *ChargerCliche*, mais ce sera nécessaire pour utiliser LiCore, comme nous allons le voir dans la partie suivante.

Cette partie ayant expliqué le concept du MSFR et les fonctionnements respectifs du code LiCore et d'Alices®, nous pouvons désormais aborder les modifications apportées et le développement de l'encapsulation au sein de la plateforme.

II. Encapsulation du code LiCore dans Alices®

L'encapsulation doit passer par une adaptation du code LiCore : en l'état, il n'a pas la même ''philosophie'' que la conception modulaire de la plateforme qui sépare chaque fonctionnalité. Dans LiCore, simulation, IHM et représentation graphique sont intrinsèquement liées.

Il a donc fallu commencer par découpler ces différents aspects, puis faire appel à cette version modifiée à l'aide d'une bibliothèque C++ dédiée.

NB : Les méthodes servant uniquement à l'import/export des valeurs des variables d'échange ne sont pas indiquées dans les tableaux de cette section. Elles seront détaillées dans la section II.2)b.

1) Modifications apportées à LiCore

a) LiCoreCouple

La classe *LiCoreCouple*, représenté dans le Tableau 8, a pour rôle de remplacer le *Bourreau* mentionné en partie I. Jusqu'ici, le processus de simulation porté par *LeGrandGestionnaire* était lancé depuis la classe qui gérait l'IHM d'initialisation. Dans l'optique du découplage, les lancements des deux processus (simulation et visualisation) ont été déplacés au sein de *LiCoreCouple*.

LiCoreCouple	
LePetitGestionnaire	**simul**
LeGestionnaireGraphique	**visu**
boolean	**logOn**
void	**main**
void	**init**
void	**doStep**
void	**terminate**
LePetitGestionnaire	**config_simul**

Tableau 8 : Classe Java LiCoreCouple

Les méthodes *init*, *doStep* et *terminate* font écho aux méthodes obligatoires d'un module Nv3 ; elles seront appelées par le code C++. La méthode main sert à lancer la simulation sans module afin de contrôler que les modifications n'altèrent pas le fonctionnement de LiCore. Dans cette optique, on dispose également du booléen *logOn* servant à indiquer si l'on souhaite produire un fichier log ou non.

b) LePetitGestionnaire

Cette classe est basée sur *LeGrandGestionnaire* auquel ont été retirées les fonctionnalités d'affichage. De plus la boucle de simulation a également été modifiée : au lieu d'une condition sur le temps écoulé, on utilise un booléen *end* qui sera modifié à l'appel de la méthode *Terminaison* du module :

```
while(temps<temps_max){
        calculer un pas
}
```

```
while(!end){
   if(top){
      calculer un pas
   }
}
```

Le *top* de cette structure de boucle est fixé sur *true* par Alices à chaque pas de temps via la méthode *make_a_step*. Cette structure de boucle utilisant *done* et *top* permet d'éviter que le module C++, beaucoup plus rapide, n'appelle *make_a_step* un grand nombre de fois alors que le calcul du pas précédent n'est pas terminé.

LePetitGestionnaire			
ArrayList<Reacteur>	**reacteurs**	Variables d'import	
boolean	**done**	**double**	**deltaRho**
boolean	**top**	**double**	**debit_coeur_import**
boolean	**end**	**double**	**debit_interm_import**
Variables d'export et d'état		**double**	**temp_interm_import**
ArrayList<AffParam>	**variables**	**double**	**puiss_extraite_import**
double[]	**cliche**	Exportation champs	scalaire
int	**nbreEtats**	**ChampScalaire[]**	**champs**
void	**do_your_jog**		
void	**make_a_step**		
void	**endSimul**		
boolean	**isDone**		
void	**prepareExportTemperatures**		

Tableau 9: Classe Java LePetitGestionnaire

Le booléen *done* et la méthode *isDone* permettent de signaler à Alices® la fin du calcul d'un pas de temps, et la classe d'origine a également été modifiée pour donner accès aux différentes grandeurs qui seront les variables d'échange du module.

Les variables d'échange au sein d'Alices® ne peuvent être que de type réel ou entier, éventuellement listées. Donc pour exporter les champs de scalaires du module de calcul vers module d'affichage le type *ArrayList<ArrayList<ArrayList<ArrayList<Double>>>>* vu plus haut ne convient pas : il faut le mettre sous forme de liste unidimensionnelle. La classe ChampScalaire permet d'effectuer ces conversions facilement.

Rappel : Pour ne pas surcharger ce paragraphe, les méthodes dédiées à la gestion des variables d'échange ne sont pas affichées ici. Cet aspect est traité en section 3) de cette partie.

c) *LeGestionnaireGraphique*

Contrairement au *GrandGestionnaire*, *LePetitGestionnaire* ne prend pas en charge l'affichage du MSFR. Cette fonctionnalité est désormais assurée par une nouvelle classe : *LeGestionnaireGraphique*. Il contient les différents éléments permettant la visualisation comme décrite dans la partie précédente : fenêtre, panneau et dessin. Sur le même principe que *done* dans *LePetitGestionnaire*, *dessinFini* est un booléen permettant de signaler la fin du pas, et *end* remplit le même rôle que son homonyme.

LeGestionnaireGraphique	
ArrayList<Reacteur>	**reacteurs**
JPanel	**fond**
JFrame	**fenetre**
PanDessinReact	**dessin**
boolean	**dessinFini**
boolean	**end**
Champ à afficher	
ChampScalaire	**champScalaire**
void	**init**
void	**doStep**
void	**terminate**
void	**set_mode_aff**

Tableau 10 : Classe Java LeGestionnaireGraphique

La méthode *set_mode_aff* permet de choisir de visualiser le champ de température ou le champ de puissance au sein du réacteur. Enfin, on retrouve les méthodes *init*, *doStep* et *terminate* analogues à celles de *LiCoreCouple.*

2) Dialogue C++/Java : la bibliothèque JNI

a) Importation et utilisation de la JVM et des objets Java

- Introduction à la JNI

La bibliothèque JNI, pour « Java Native Interface » offre la possibilité de faire appel à du code Java à partir d'un programme C++. L'utilisation se fait en trois temps.

Il faut tout d'abord encapsuler une instance de la machine virtuelle Java, ou JVM, qui permettra de faire fonctionner les objets et méthodes Java. On crée alors un *environnement JNI* contenant les méthodes nécessaires par la suite.

La deuxième étape consiste à importer les classes, méthodes et champs que l'on souhaite pouvoir utiliser. Concrètement, cela consiste à appeler une méthode adaptée de l'environnement JNI en lui indiquant le nom de l'objet Java souhaité, vers lequel sera alors créé un pointeur (en programmation, un pointeur est une variable indiquant l'adresse mémoire d'une autre variable).

Enfin vient l'utilisation à proprement parler. En utilisant le pointeur de l'objet que l'on veut utiliser et la méthode JNI adaptée on peut appeler une méthode, consulter ou imposer la valeur d'un champ. Le tableau 11 illustre les différentes correspondances.

Objet Java	Classe C++ correspondante	Méthodes JNI
Classe	**jclass**	**FindClass(**name**)**
Méthode	**jmethod**	**GetMethodID(**jclass**,**name**,**signature**)** **GetStaticMethodID(**jclass**,**name**,**signature**)** **Call<type>Method(**jobject**,**jmethodID**,…)** **CallStatic<type>Method(**jclass**,**jmethodID**,…)**
Attribut	**jfield**	**GetFieldID(**jclass,name**,**signature**)** **Get<type>Field(**jobject,jfieldID**)** **Set<type>Field(**jobject,jfieldID,value**)** **GetStatic<type>Field(**jclass,jfieldID**)** **SetStatic<type>Field(**jclass,jfieldID,value**)**

Tableau 11: Correspondances Java/C++/JNI

La signature est une chaîne de caractères permettant à la JNI de savoir quel est le type d'un champ, et les types de paramètres et retour d'une méthode donnée. Les correspondances caractères/type Java sont fournies en annexe.

b) Classes d'utilisation des objets Java

Afin d'alléger le code du module, on procède à l'élaboration de classe dédiées à l'importation et l'utilisation des objets Java (Tableaux 12 à 14).

JavaClass	
void	**loadClass**
void	**loadMethods**
void	**loadFields**
void	**dealocate**
char*	**mName**
jclass	**mClass**

Tableau 12 : Classes C++ d'utilisation des classes Java

Cette classe *JavaClass* contient deux méthodes qualifiées d'« abstraites » : *loadMethods* et *loadFields*. Cela signifie qu'elles doivent être redéfinies au sein de chaque classe héritant de *JavaClass.* Nous verrons dans la section suivante que cela permet d'avoir recours aux listes d'initialisations pour créer des images C++ de nos classe Java, donc de les instancier avec un minimum de syntaxe.

JavaMethod	
void	**load**
T	**call**
char*	**mName**
char*	**mSignature**
jmethodID	**mID**

Tableau 13 : Classe C++ d'utilisation des méthodes Java

Les *JavaMethod* sont dans un premier temps instanciées à partir de leurs nom et signature lors de la construction d'une *JavaClass*. C'est dans un second temps qu'elles sont réellement importées, à l'appel de leur méthode *load* qui initialise leur identifiant.

La méthode *call* est une méthode « template » qui permet d'avoir une unique fonction qui englobe toutes les méthodes d'appel de la forme *Call<type>Method*. On utilise la signature de la *jmethod* pour déterminer la méthode à utiliser. Ainsi toutes les méthodes Java sont appelées de la même façon, qu'elle soit statique ou non, et quel que soit son type de retour.

JavaField	
void	**load**
void	**setValue**
T	**getValue**
char*	**mName**
char*	**mSignature**
jfieldID	**mID**

Tableau 14 : Classe C++ d'utilisation des champs Java

Enfin, les *JavaField* fonctionnent sur le même principe d'importation en deux temps, et *getValue* et *setValue* permettent de regrouper en deux méthodes toutes les fonctions de consultation et d'attribution de valeur de champ.

3) Module de simulation

a) Classes C++ images des classes de LiCore

- LiCoreCouple

La classe *LiCoreCouple* hérite de *JavaClass*, donc comme indiqué plus haut elle redéfinit les méthodes *loadMethods* et *loadFields*. Cette classe reproduit la structure de la classe Java *LiCoreCouple* dans sa quasi-totalité : on retrouve les méthodes d'initialisation, de calcul d'un pas de temps, et de finalisation de la simulation.

LiCoreCouple	
void	**loadMethods**
void	**loadFields**
PetitGestionnaire	**mPetitGestionnaire**
JavaStaticMethod	**mInit**
JavaStaticMethod	**mDoStep**
JavaStaticMethod	**mTerminate**

Tableau 15 : Classe C++ LiCoreCouple

- PetitGestionnaire

En plus de l'accès au booléen nécessaire à la simulation, cette classe contient toutes les *JavaMethod* nécessaires à la communication des variables d'import et d'état entre le module C++ et LiCore Java.

PetitGestionnaire			
void	**loadMethods**		
void	**loadFields**		
Gestion de la simulation		Gestion des champs scalaires	
JavaStaticMethod	**mIsDone**	**JavaStaticMethod**	**mGetTailleTotale**
JavaStaticField	**mNbreReact_F**	**JavaStaticMethod**	**mGetElmtChamp**
JavaStaticField	**mDone**	**JavaStaticMethod**	**mSetElmtChamp**
Gestion de clichés		Gestion des variables d'import	
JavaStaticMethod	**mSetElmtCliche**	**int**	**mNbreVarImport**
JavaStaticMethod	**mGetElmtCliche**	**JavaStaticField***	**mListImport[5]**
JavaStaticMethod	**mPrendreCliche**	**JavaStaticField**	**mDeltaRho**
JavaStaticMethod	**mSetCliche**	**JavaStaticField**	**mDebitCoeur**
Importation des variables d'export		**JavaStaticField**	**mDebitInterm**
JavaStaticMethod	**mGetVarExport**	**JavaStaticField**	**mTempInterm**
		JavaStaticField	**mPuissExtraite**

Tableau 16 : Classe C++ PetitGestionnaire

mPrendreCliche permet d'ordonner à LiCore de stocker les valeurs des variables nécessaires à la prise d'un cliché. Ces valeurs sont ensuite récupérées une à une via *getElmtCliche*. Lors de l'opération inverse (le chargement d'un cliché) les valeurs sont passées une à une du module vers

LiCore via *setElmtCliche*. Une fois toutes les valeurs mises à jour dans le champ *cliche* du *PetitGestionnaire* de LiCore, et dans toutes les instance de *CelluleFluide*, elles sont appliquées en faisant appel à la méthode *mSetCliche*. Les valeurs contenues dans les champs scalaires font également partie des variables d'état, elles sont gérées sur le même principe, terme à terme.

Enfin, on instancie un *JavaStaticField* pour chaque variable d'import afin de pouvoir fixer leurs valeurs comme ordonnées par l'utilisateur via l'IHM de pilotage. Un extrait du code de cette classe est disponible en Annexe 2.

- Variables

Cette *JavaClass* est image de la classe *AffParam* de LiCore, qui servait initialement au stockage des grandeurs affichées dans des graphes. Elle contient désormais des attributs supplémentaires, correspondant aux variables d'export et d'état (sauf les champs scalaires), voir le Tableau 17.

Variables			
void	**loadMethods**		
void	**loadFields**		
JavaField	**getVarExport**		
JavaField	**getVarEtat**		
JavaField*	**mListVarExport[6]**	**JavaField***	**mListVarEtat[5]**
JavaField	**mPuissance**	**JavaField**	**mPuissExtraite**
JavaField	**mReac_Beta**	**JavaField**	**mPopulation**
JavaField	**mPrecurseurs**	**JavaField**	**mPopulationOld**
JavaField	**mTempMasse**	**JavaField**	**mDebitCoeur**
JavaField	**mTempMax**	**JavaField**	**mDebitInterm**
JavaField	**mTempMin**		

Tableau 17 : Classe C++ Variables

Les deux méthodes *getVarExport* et *getVarEtat* permettent d'accéder aux valeurs des différentes variables, en particulier pour les mettre à jour à chaque pas de temps.

b) Classe module_licore

C'est cette classe qui définit le module Nv3 comme décrit dans la première partie, elle comprend donc les méthodes de base *Nouveau Initialisation*, *Traitement* et *Terminaison*. Elle redéfinit de plus les méthodes d'enregistrement et de chargement des clichés afin d'assurer la communication des valeurs des variables entre le module Nv3 et LiCore. Le tableau 18 montre les méthodes supplémentaires et les attributs de cette classe.

module_licore	
Méthodes auxiliaires	
vector<T>	creerListVarPtr
void	declarerVariables
SruString	nomDeVariable
void	envoiImports
LiCoreCouple	mLiCoreCouple_C
PetitGestionnaire	mGestionnaire_C
Variables	mVariables_C
jclass	mLiCoreCouple_J
jclass	mGestionnaire_J
jclass	mVariables_J
Champs scalaires	
int	mTailleTotalChamp
int	mNbreDeChamps
vector<vector<double*>>	mListChamps
Variables d'export	
int	mNbreExport
vector<double*>	mListeExport
Variables d'état	
int	mNbreEtat
vector<double*>	mListeEtat
Variables d'import	
int	mNbreImports
vector<double*>	mListImports

Tableau 18 : Classe C++ module_licore

Pour être déclarée, chaque variables doit avoir un nom d'échange « jeton1 :jeton2 :jeton3 » qui est construit à l'appel de la méthode *nomDeVariable*, et on stocke les pointeurs de ces variables dans des objets *vector*, équivalent C++ des *ArrayList* Java. Ces *vector* sont créés via la méthode template *creerListVarPtr* qui crée un vecteur du type voulu (*int** ou *double**)

La méthode *Initialisation* crée la géométrie du réacteur simulés via la méthode *init* de *LiCoreCouple* et déclare toutes les variables d'échange, et *Terminaison* fait appel à *terminate* de *LicoreCouple*. Le déroulement de *Traitement* quant à lui contient plusieurs étapes :

- Passage de la valeur booléenne *done* du *PetitGestionnaire* sur false
- Appel de la méthode *doStep* de *LiCoreCouple*
- Attente de la fin du calcul, signalé par le retour de la valeur de *done* sur true
- Mise à jour des variables exportées

Le code de cette méthodes est en Annexe 3.

Il est désormais possible de lancer une simulation LiCore dans une configuration Alices®, mais rappelons que le but de ce projet est de pouvoir étudier le comportement d'un MSFR en le pilotant en temps réel, et non pas en prévoyant la totalité du scénario comme c'est le cas avec le code LiCore seul. Il faut donc disposer d'un outil permettant de mettre en place les modifications au moment souhaité durant la simulation, et c'est ici qu'intervient le concept de module de contrôle.

III. Pilotage et résultats

Un module de contrôle permet de décider des valeurs des variables d'import du module de calcul, et pour que l'utilisateur puisse interagir facilement, il convient de construire une IHM qui sera branchée sur ce module de contrôle.

1) Construction du pilotage

a) Document Nv2 du module de contrôle

On souhaite pouvoir réaliser des variations sur les variables d'import du module de calcul, il faut donc plusieurs paramètres : une valeur de départ, de fin, une durée, et un type de variation. On se propose d'implémenter trois types de variations, présents dans LiCore : directe, linéaire et exponentielle.

Puisque chaque type de variation doit être possible pour toutes les variables, on procède à la construction d'un macro objet qui retourne la valeur à l'instant t en fonction des paramètres cités ci-dessus.

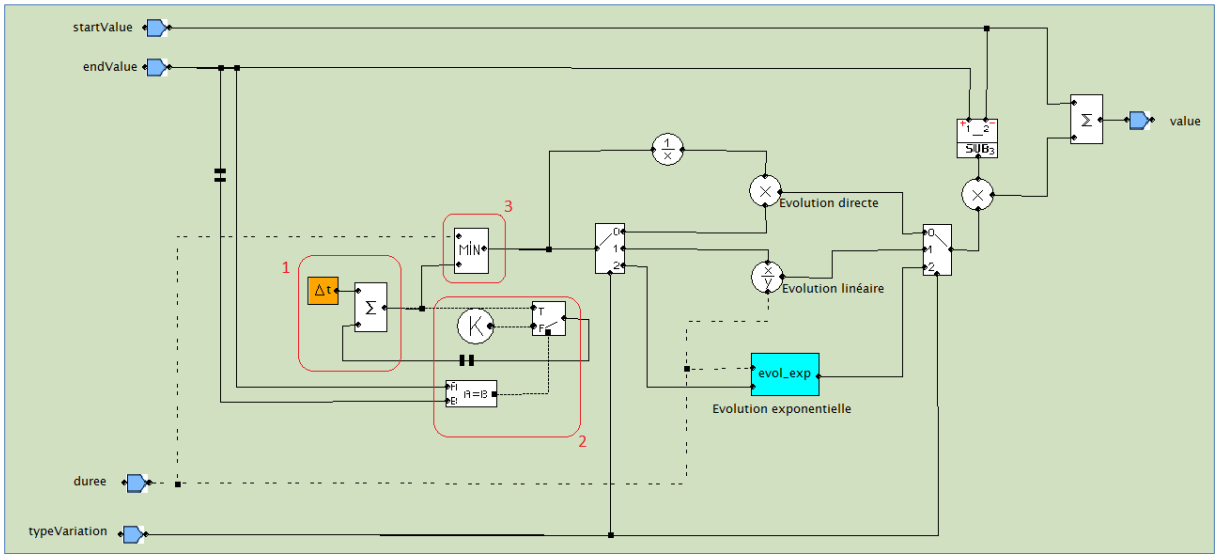


Figure 13 : Document du macro objet de calcul de variation

Comme représenté sur la Figure 13, il dispose de quatre entrées (à gauche) correspondant aux paramètres, et retourne la valeur à l'instant donné (à droite). Pour cela, le temps est itéré à chaque pas (zone 1) en partant de 0, réinitialisé à chaque fois que la consigne de valeur finale est modifiée (zone 2). Une fois que le temps écoulé a dépassé la durée, la sortie reste constante car on calcule *value* à l'instant *t=durée* (zone 3).

Ainsi en utilisant ce macro objet on construit facilement le document Nv2 du module de contrôle, dont la Figure 14 illustre une partie.

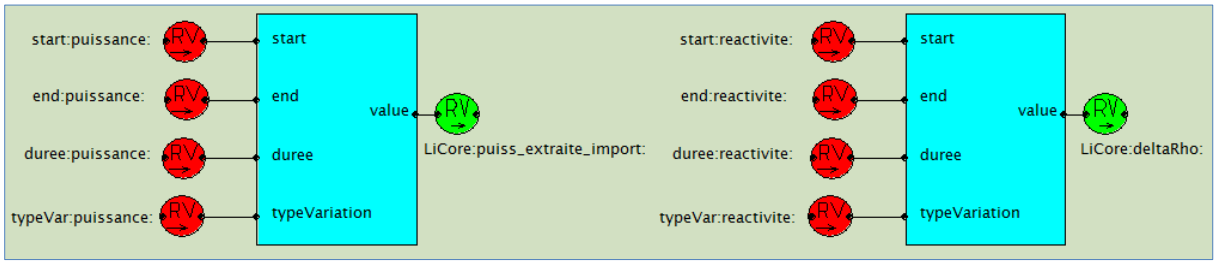


Figure 14 : Extrait du document Nv2 du module de contrôle

Ce module importe les variables ayant comme premier jeton « start » ; « end » ; « duree » et « typeVar » fournis par l'IHM, et exporte les variable importées par le module de calcul.

b) Document et dessin Nv2 de l'IHM

L'IHM de pilotage est composée de deux éléments : son dessin et son document. Le dessin peut être fait à partir de deux prototypes : curseurs et boutons, et le document assure deux rôles.

Le premier est d'assurer la cohérence entre valeurs exportées et apparence de l'IHM (positions des curseurs, sélection des boutons), et le second est d'exporter les consignes de l'utilisateur au module de contrôle.

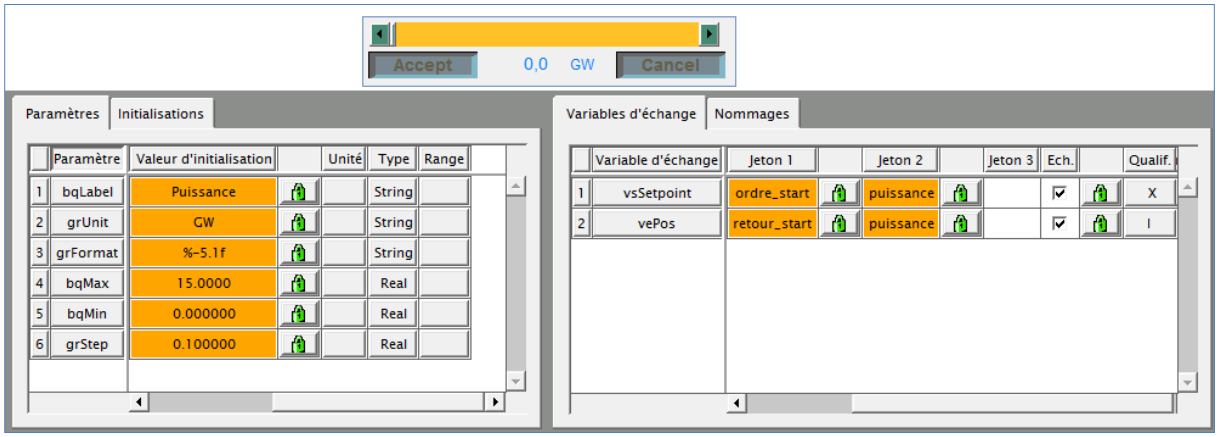


Figure 15 : Fiches Nv2 du prototype de curseur

La Figure 15 illustre le paramétrage dur curseur permettant de fixer la consigne de valeur de départ pour une évolution de puissance extraite. La variable *vsSetPoint* contient la consigne fixée par l'utilisateur et est exportée lorsque l'utilisateur clique sur « Accept », et *vePos* est la variable imposant la position du curseur. Les prototypes de groupes de deux ou trois boutons présentent une structure identique pour leurs variables d'échange.

2) Résultats

a) Configuration de simulation

La Figure 16 illustre la configuration de simulation à l'issu de ce stage. On y retrouve les modules exposés plus haut, ainsi que *GestionSimu,* dont le rôle est principalement de gérer le pas de temps de la simulation.

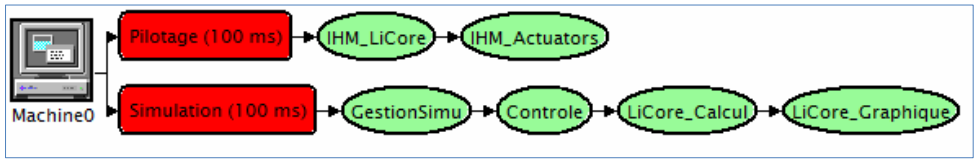


Figure 16 : Configuration finale

Les classes d'utilisation d'objets Java exposées en partie II, ainsi qu'une classe dédiée à l'environnement JNI sont utilisée par les deux modules LiCore_Calcul et LiCore_Graphique. Pour éviter d'avoir des fichiers en doublon, ce qui pourrait poser problème lors de mises à jour ultérieures, elles sont compilées dans un fichier à part nommé LiCore_Common. N'étant pas un module à proprement parler, il n'apparaît pas sur la configuration.

b) *Aspect général*

La Figure 18 est une capture de l'interface de pilotage dans sa dernière version. Dans chaque cadre consacré à une variable, le curseur du haut règle la valeur de départ de l'évolution, et celui en bas à gauche gère la valeur finale. Ainsi, l'évolution commence lors du clic sur le bouton « Valider » de ce curseur. On dispose par ailleurs de boutons permettant de choisir le type de variation, et d'un troisième curseur pour la durée.

Cette IHM permet également de choisir à tout moment d'afficher la température ou la puissance au sein du réacteur, et le moniteur d'Alices® offre la possibilité d'afficher des graphiques pour n'importe quelle variable d'échange de la configuration. On s'intéressera tout particulièrement aux variables d'export du module de calcul : températures maximale, minimale et moyenne ; population de précurseurs, écart à la criticité prompte. Enfin, le pilotage permet de lancer une variation, ou bien toutes à la fois.

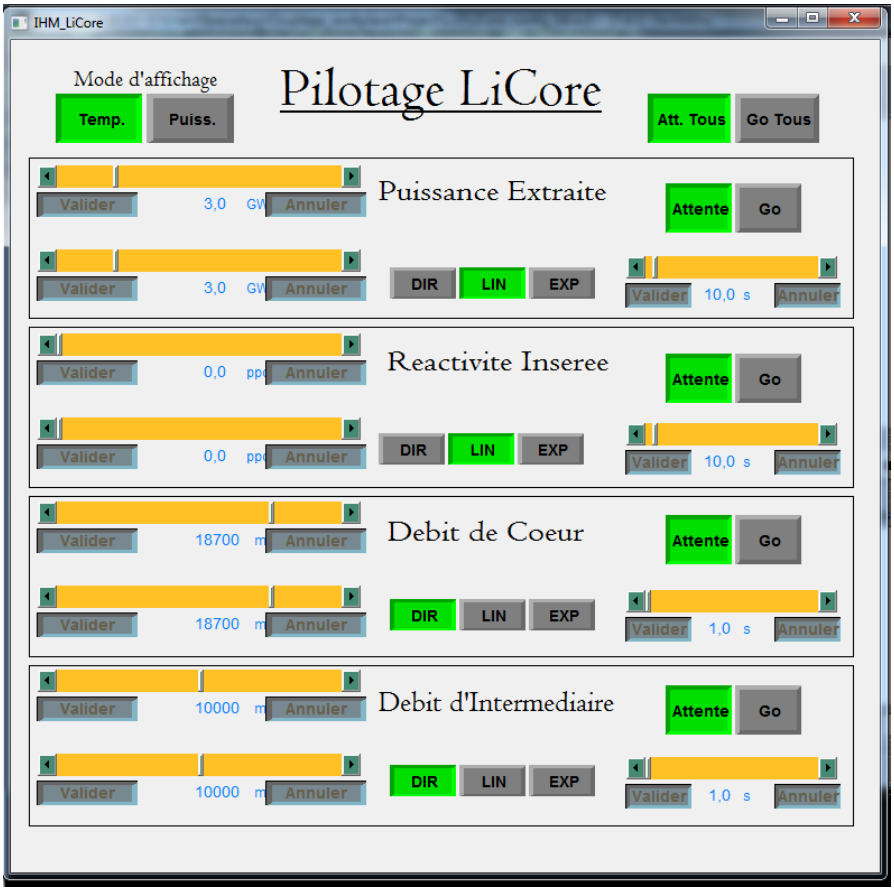


Figure 17 : IHM de pilotage

c) *Etude de transitoire*

On se propose de procéder à une étude comparable à celle illustrée dans la thèse d'Axel Laureau [3], ce qui permettra de confirmer que le calcul LiCore n'a pas été altéré lors de l'adaptation en module Alices®.

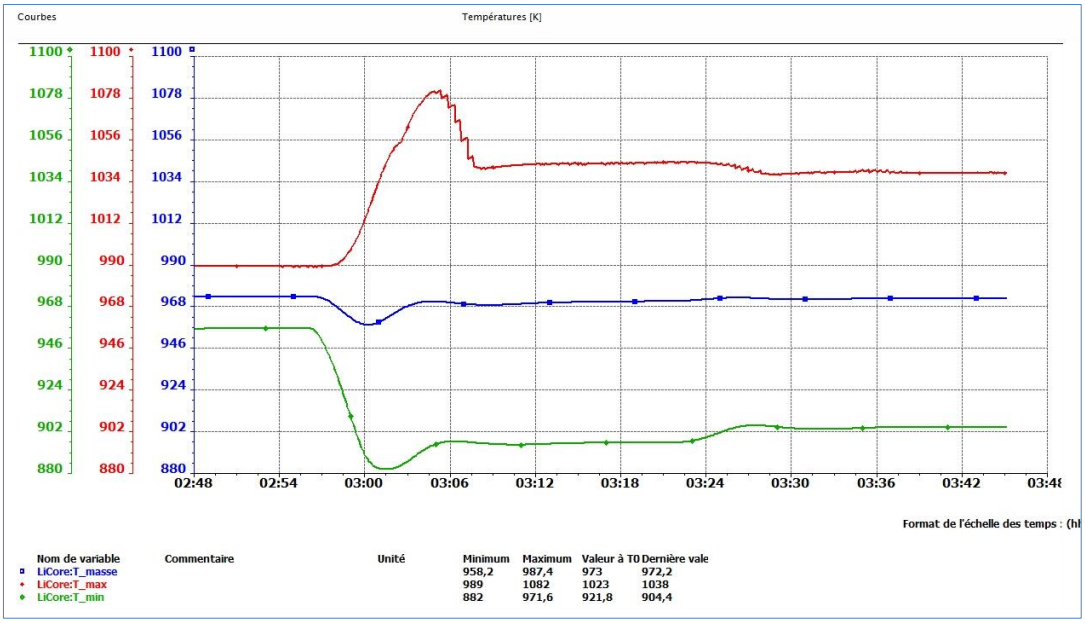


Figure 18 : Montée en puissance de 1GW à 4GW

Après une mise à l'équilibre à une puissance extraite de 1GW, une montée instantanée de la puissance extraite à 4GW donne l'évolution de températures illustrée en Figure 17. On retrouve sur ce graphe une variation d'environ 20K sur la température moyenne (courbe bleue), et un équilibre atteint une trentaine de secondes après la montée en puissance, ce qui correspond aux résultats obtenus auparavant ([3], chap.5, Fig. 5.2).

Alices® permet d'exporter les courbes obtenues sous forme de fichier .csv et donc d'exploiter les données obtenues à l'aide d'un tableur. Ainsi en effectuant cette montée de puissance extraite pour différentes puissances de départ on obtient le graphe de comparaison présenté en Figure 18.

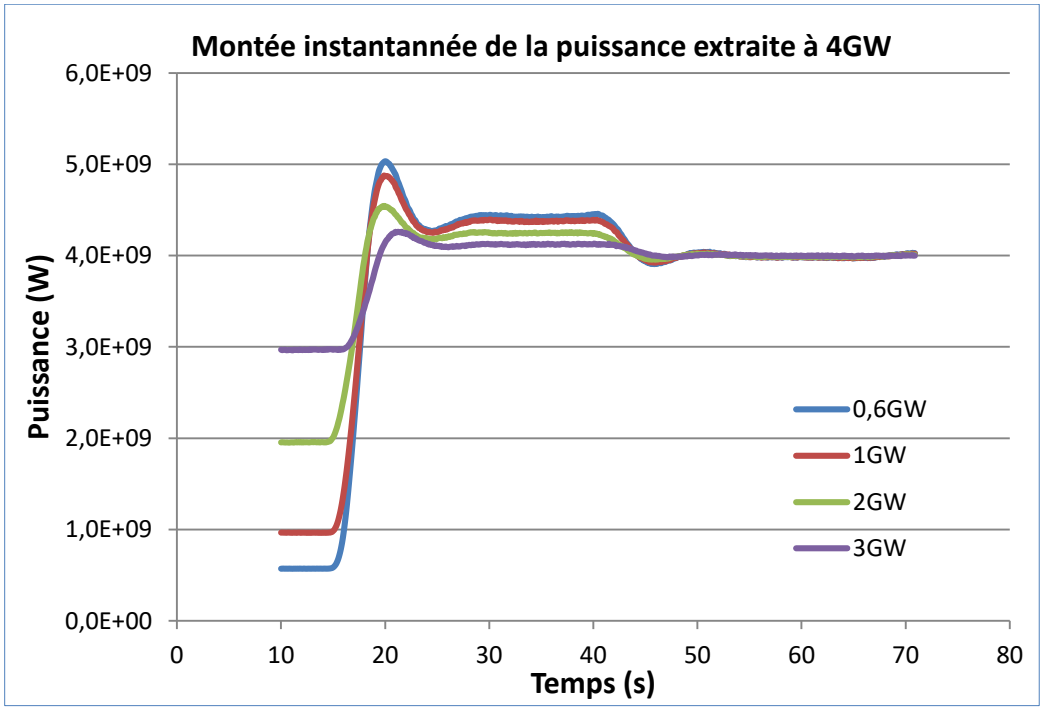


Figure 19 : Montée en puissance instantanée pour différentes puissance de départ

Là encore, on retrouve une évolution qui commence par des oscillations, la puissance de consigne étant d'autant plus dépassée que la puissance de départ est basse. Cet effet avait également été constaté avec LiCore seul ([3], chap. 5, Fig. 5.5).

Le dépassement de la puissance est dû au refroidissement rapide du combustible : la diminution de l'effet Doppler et l'augmentation de la densité augmente fortement les sections efficaces, et donc la puissance générée. Ensuite l'effet s'inverse : cette hausse de puissance a augmenté la température de telle façon que les sections efficaces chutent, diminuant la puissance. A cela s'ajoute les effets de circulation du sel, lui-même caractérisé par sa propre constante de temps. L'amplitude des oscillations diminuant, la puissance finit par se stabiliser.

Ainsi sous Alices® on obtient les mêmes résultats que LiCore seul, tout en étant capable de piloter le cœur en temps réel. Des améliorations pourraient être apportées, notamment sur la gestion de cliché, mais l'objectif fixé pour ce stage est finalement atteint.

Conclusion

Ce rapport a donc eu pour objectif d'exposer les différentes étapes de l'adaptation du code LiCore en modules Alices®. Après avoir pris en main les différents outils de développement il a fallu modifier LiCore afin que sa structure permette une communication aisée avec deux modules dédiés à deux tâches distinctes : la simulation d'une part et la visualisation d'autre part. Ensuite il a fallu développer la fonctionnalité que l'on souhaitait apporter à LiCore : le pilotage. Ce fut réalisé via un module de contrôle transmettant les consignes de l'utilisateur depuis une IHM vers le module de calcul.

Ce stage a démontré la faisabilité d'inclure des codes développés au CNRS dans l'environnement Alices®. On dispose désormais de la base de ce qui pourra devenir un simulateur complet pour une centrale nucléaire équipée d'un MSFR. La structure actuelle pourra être complétée en utilisant les objets présents dans les bibliothèques d'Alice® (autres modules de contrôle, descriptions plus précises des circuits intermédiaire et de conversion de l'énergie avec ajout de turbines, gestion de pannes…), ou bien des codes de calcul tiers qui seront encapsulés sur le même principe que LiCore.

Références

[1] Site du LPSC : http://lpsc.in2p3.fr/index.php/fr/

[2] Site de Corys : http://www.corys.com/fr

[3] M. Allibert, M. Aufiero, M. Brovchenko, S. Delpech, V. Ghetta, D. Heuer, A. Laureau, E. Merle-Lucotte, *"Chapter 7 - Molten Salt Fast Reactors"*, Handbook of Generation IV Nuclear Reactors, Woodhead Publishing Series in Energy (2015)

[4] Allibert M., Gérardin D., Heuer D., Huffer E., Laureau A., Merle E., S. Beils, A. Cammi, B. Carluec, S. Delpech, A. Gerber, E. Girardi, J. Krepel, D. Lathouwers, D. Lecarpentier, S. Lorenzi, L. Luzzi, S. Poumerouly, M. Ricotti, V. Tiberi, , "*Description of initial reference design and identification of safety aspects of the MSFR*" Groupe de Travail 1, livrable 1.1, SAMOFAR European H2020 Project, Contract number: 661891 (2017)

[5] Axel LAUREAU, "*Développement de modèles neutroniques pour le couplage thermohydraulique du MSFR et le calcul de paramètres cinétiques effectifs*», Thèse de doctorat, Université Grenoble Alpes, France (2015)

[6] Allibert M., Gérardin D., Heuer D., Huffer E., Laureau A., Merle E., S. Beils, A. Cammi, B. Carluec, S. Delpech, A. Gerber, E. Girardi, J. Krepel, D. Lathouwers, D. Lecarpentier, S. Lorenzi, L. Luzzi, S. Poumerouly, M. Ricotti, V. Tiberi, , "*Description of initial reference design and identification of safety aspects of the MSFR*" Groupe de Travail 1, livrable 1.1, SAMOFAR European H2020 Project, Contract number: 661891 (2017)

[7] OpenClassroom, cours « Apprenez à programmer en Java » https://openclassrooms.com

[8] Documentation Corys : « ALICES® : Guide de prise en main »

[9] Site Code Project : article « Calling Java from C++ with JNI » https://www.codeproject.com/Articles/993067/Calling-Java-from-Cplusplus-with-JNI

[10] Documentation Oracle sur la JNI : https://docs.oracle.com/javase/7/docs/technotes/guides/jni/spec/jniTOC.html

Annexes

- **Annexe 1 : Table de correspondance signature / type Java ([10] table 3-2)**

Toutes les classes non primitives doivent être signées par leur nom complet. La signature d'une méthode est composée de la signature des paramètres entre parenthèses, suivie de celle du retour.

Exemple tiré de la documentation Oracle : la méthode

long fonction(int n, String str, int[] list)

a pour signature

"(ILjava/lang/String;[I)J"

Signature	Type Java
Z	boolean
B	byte
C	char
S	short
I	int
J	long
F	float
D	double
L	Nom complet
[	Tableau

- **Annexe 2 : Extrait de la classe C++ PetitGestionnaire :**

```
class PetitGestionnaire : public JavaClass {
        public:
        PetitGestionnaire()
                : JavaClass             ("Launcher/LePetitGestionnaire"     )
                /*Methode de tag de la fin du calcul d'un pas */
                ,mIsDone                ("isDone"                   ,"()Z"     )
                /* Prise de cliché */
                ,mPrendreCliche         ("prendreCliche"            ,"()V"     )
                /*Gestion des champs scalaires */
                ,mGetElmtChamp          ("getElmtChamp"             ,"(II)D"   )
                ,mSetElmtChamp          ("setElmtChamp"             ,"(IID)V" )
                /*Variables d'imports */
                ,mDeltaRho              ("deltaRho"                 ,"D"       )
                ,mPuissExtraite         ("puiss_extraite_import"    ,"D"       )
                        (...)
                {}

        /** Importation des methodes */
        virtual void            loadMethods             ();
        /** Importation des attributs */
        virtual void            loadFields              ();
                        (...)
        private:
        JavaStaticMethod        mIsDone         ;
        JavaStaticMethod        mPrendreCliche  ;
        JavaStaticMethod        mGetElmtChamp   ;
        JavaStaticMethod        mSetElmtChamp   ;
        JavaStaticField         mDeltaRho       ;
        JavaStaticField         mPuissExtraite  ;
                        (...)
};
```

Lors de la déclaration d'une classe, le nom doit être sous la forme « package/classe », quant aux méthodes et attributs, on voit que leur déclaration est simplifiée par les classe *JavaStaticMethod* et *JavaStaticField*. Ces dernières sont héritées respectivement de *JavaMethod* et *JavaField,* et ont le même comportement.

Les *JavaClass LiCoreCouple*, *Variables*, *GestionnaireGraphique*, se présentent sous la même forme.

- **Annexe 3 : Méthode Traitement du module de calcul**

```
void module_licore::Traitement() {
	// Pas de temps fourni par Alices
	int timeStep = TimeStepDT() ;
	// Récupération des variables d'import par Java
	envoiImport() ;
	// Passage de la variable 'done' sur false
	mGestionnaire_C.done().setValue<jboolean>(mGestionnaire_J,false) ;
	// Lancement du calcul
	mLicoreCouple_C.step().call<void>(mLicoreCouple_J,timeStep) ;
	// Attente
	jboolean done ;
	while(true){
		Sleep(10);
		done = mGestionnaire_C.isDone().call<jboolean>(mGestionnaire_J);
		if(done)
			break ;
	}
	// Mise à jour des variables exportées
	for(int i=0 ; i<mNbreExport ; i++)
		*(mListeExport[i]) = (double)mGestionnaire_C.getVarExport().call<jdouble>(mGestionnaire_J,i) ;
}
```

On retrouve notamment la méthode *template call* mentionnée dans le corps du rapport, permettant d'appeler n'importe quelle méthode Java à l'aide d'une unique méthode. La seule contrainte est d'indiquer le type de retour entre les token.

Résumé

Dans le cadre du développement du *Molten Salt Fast Reactor* (MSFR), concept de réacteur à combustible liquide, le LPSC de Grenoble dispose entre autres d'un code système nommé LiCore permettant d'évaluer rapidement des ordres de grandeur caractéristiques de ce système. Il présente toutefois une légère contrainte : la totalité du scenario de simulation doit être prévu à l'avance. On souhaite pouvoir procéder à un pilotage en temps réel. Ce rapport expose comment le code LiCore a été adapté et encapsulé afin de pouvoir être utilisé sur la plateforme de simulation Alices® développée par l'entreprise Corys.

On commence par se familiarisé avec le concept du MSFR et les fonctionnements respectifs du code LiCore et d'Alices®, afin d'aborder dans une deuxième partie les modifications apportées au code LiCore et le développement des outils permettant l'encapsulation. En dernière partie sont exposés la construction d'une interface de pilotage et le résultat obtenu à l'issu de ce stage.

Abstract

To work on the *Molten Salt Fast Reactor* (MSFR), a concept of liquid fuel powered nuclear reactor, the LPSC (Grenoble, France) has developed a system code named LiCore. It can quickly evaluate typical values and behaviors for such system, but it also has a restriction : the whole scenario must be planed before launching the computation. One would like to be able to drive the simulation in real time. This report exposes how LiCore was adapted and encapsulated to be used on Alices®, an integrated simulation toolset developed by the company Corys.

The MSFR concept and the respective structures of LiCore and Alices® are explained in the first part. Then the second one gives details concerning the modifications brought to LiCore, and the development of the tools used to encapsulate it. The graphical user interface development and the result at the end of this internship are exposed in the last part.