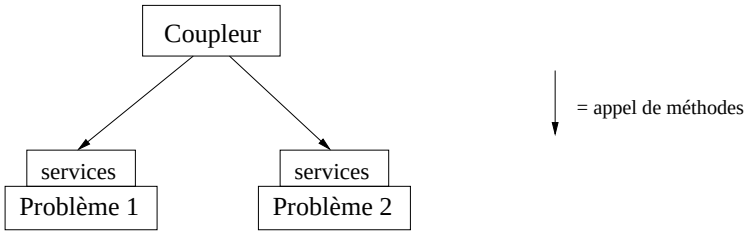
Table des matières

1 Introduction

Les réalisations de couplages multi-échelles ou multi-disciplines se multiplient [15, 5, 17, 18, 12]. Pourtant il n'y a pas à ce jour de standardisation qui permette de capitaliser les acquis de ces couplages dans le domaine du nucléaire, même si certaines briques ont été développées (coupleurs ISAS, Calcium). Intuitivement, le couplage peut s'organiser selon le schéma ci-dessous, où les problèmes présentent des services que le coupleur peut appeler (schéma maître-esclave).



Il existe également des couplages de codes sans maître, avec échanges de données directement entre les codes. Cette approche impose à chaque code de connaître l'existence et les caractéristiques de ceux avec lesquels il est

couplé, notamment quand et sous quelle forme ils émettent ou reçoivent des données. Elle est donc moins souple et plus spécifique à des cas particulier.

La mise en oeuvre des couplages envisagés dans le cadre de la plateforme Neptune doit rester la plus simple et la plus accessible possible.

Actuellement, pour coupler deux codes, et en supposant que ces deux codes présentent une interface (en python par exemple), l'utilisateur doit faire face à de nombreuses difficultés qui ne relèvent pas directement de son travail.

Tout d'abord, chaque code développe une interface spécifique, au plus proche de son fonctionnement interne mais pas forcément très lisible ni très adaptée à l'utilisation par un tiers en vue du couplage. L'utilisateur doit s'approprier ces deux interfaces, et comprendre à quoi sert chaque méthode.

Mais surtout, le couplage nécessite de lire et d'écrire des informations dans chacun des codes. Des méthodes sont logiquement disponibles dans ce but. Même si l'interface de chaque code est bien documentée et que l'utilisateur investit le temps nécessaire à se l'approprier, il sera souvent difficile de savoir quels sont précisément les effets des méthodes de lecture et d'écriture de données dans les codes. Or, la plupart des applications relevant du couplage impliquent des transitoires, il est donc indispensable de maitriser précisément la dépendance en temps des résultats.

Quand doit-on lire une information ? Cela peut être indifférent, ou bien il est nécessaire d'attendre la fin du pas de temps, ou l'exécution d'une méthode spéciale,... Quelle est la signification de ce qu'elle contient ? Cela peut être le champ à l'instant n, ou $n+1$, ou encore autre chose suivant la façon dont le code gère l'avance en temps. Quand le résultat va-t-il changer ? Probablement à l'appel d'une méthode qui gère le calcul sur le prochain pas de temps, mais peut-être aussi quand on impose une condition aux limites, que l'on change le matériau, que l'on fait un projection sur l'espace des solutions acceptables,... Peut-être la densité va-t-elle changer au moment où on impose la température ? Il est difficile de le prédire à la seule lecture de l'interface.

De même, quand faut-il imposer une donnée d'entrée ? Peut-être l'utilisateur croira-t-il avoir imposé une condition aux limites ou un terme source, mais celui ne sera pas pris en compte car l'appel n'a pas eu lieu au bon moment dans le déroulement du programme, ou peut-être sera-t-il pris en compte pour le pas de temps $n+1$ alors que l'utilisateur croyait que ce serait pour le pas de temps n. Là encore, difficile de le déduire de la lecture de l'interface.

Enfin, outre l'écriture des jeu de données pour chaque code, l'utilisateur doit écrire l'ensemble du déroulement en temps du couplage. Et s'il souhaite remplacer un des deux codes par un autre qui est censé résoudre le même problème avec un schéma de calcul différent, il devra réapprendre une nouvelle interface et réécrire le couplage en appelant cette interface différente. Cela est dissuasif et il abandonnera peut-être l'idée de comparer les résultats des deux codes.

Imaginons au contraire le travail de l'utilisateur dans un cas idéal. Il a à sa disposition toute une variété d'outils. Il dispose tout d'abord d'un certain nombre de codes, présentant tous la même interface d'utilisation. A partir du moment où il dispose de jeux de données pour ces codes, il peut les utiliser à sa guise via l'interface standardisée. Il a besoin de documentation sur un seul point : les champs d'entrée dont le code a besoin et les champs de sortie qu'il peut fournir. Ces informations lui sont fournies par la personne qui a écrit le jeu de données. Seule la signification physique des champs doit être décrite : la localisation spatiale et temporelle est entièrement spécifiée par le code via les méthodes de l'interface standardisée.

Il dispose également d'un certain nombre d'algorithmes de couplage déjà écrits par des numériciens, et implémentant diverses méthodes d'avance en temps, explicites ou itératives avec pas de temps coincidents ou non, voir de type Schwarz. Ces algorithmes sont capables de gérer n'importe quel code présentant l'interface standard. Le rôle de l'utilisateur se borne donc à choisir des codes, un algorithme, et à brancher les champs d'entrée et de sortie. Pour ce faire, il a accès à une bibliothèque d'interpolateurs en espace et en temps, ou d'objets plus spécifiques (modélisateurs-démodélisateurs) capables de calculs et de conversions sur les champs. Si le besoin s'en fait sentir, il peut aussi coder un nouvel interpolateur, ou même un nouvel algorithme de couplage.

La présente note a pour but de faire un premier pas dans cette direction en proposant une interface standard à l'objet problème. Elle ne traite ni de la façon de paramétrer les codes (jeux de données, interfaces homme-machine, objets technologiques), ni des coupleurs qui utiliseront cette interface, ni des interpolateurs qui seront disponibles pour l'utilisateur. Elle propose uniquement une liste de services standards qu'un problème doit fournir pour être

facilement couplé, mais surtout elle fixe des règles, des spécifications sur le comportement que doivent avoir ces méthodes et sur la façon de les appeler. Le but de ces spécifications, qui sont le plus souvent intuitives, est de permettre à l'utilisateur de travailler en sécurité, en étant sûr de l'interprétation qu'il doit avoir des entrées qu'il impose au problème et des sorties qu'il en reçoit.

Le document est organisé autour de deux grandes parties. La première définit l'ensemble des méthodes de l'interface de couplage et des spécifications qu'elles doivent respecter. L'autre définit la chronologie des appels, c'est-à-dire de quelle façon ces méthodes peuvent être appelées par l'utilisateur sans produire un effet indéterminé.

L'interface de couplage est aussi appelée API de couplage (pour Application Programming Interface).

2 Préalables

2.1 Position par rapport à l'existant

L'interface que allons présenter est le fruit de réflexions qui font suite à la note [11], avec d'importantes modifications. Mais elle résulte aussi de la confrontation avec la réalité des codes et d'une étude bibliographique sur les couplages déjà réalisés. En effet la recherche de généralité et de standardisation ne doit pas empêcher de reproduire dans le nouveau cadre ce qui était possible auparavant.

L'approche d'un couplage par appel de services proposés par les problèmes est très différente de celle utilisée dans Calcium [7], où les problèmes se déroulent séparément et échangent des données avec le coupleur via MPI. Cependant, tout ce qui est possible avec Calcium doit également être possible avec notre approche. En revanche, l'approche utilisée par Calcium demande des précautions très particulières lors du développement des codes et du coupleur pour que les appels MPI effectués par les codes d'une part, et par le coupleur d'autre part, coïncident.

L'API de couplage proposée ici autorise la reproduction de ce qui a déjà été réalisé dans le cadre de couplages via ISAS [6]. Par rapport à cette approche, elle apporte à la fois plus de généralité (tous les problèmes présentent les mêmes méthodes) et un grain de couplage plus fin (bouclages possibles à l'intérieur ou à l'extérieur des pas des temps). Cela autorise le codage de coupleurs utilisant des algorithmes complexes et efficaces, qui soient réutilisables pour plusieurs applications de couplage. Elle permet en outre un partage des tâches clair en évitant par exemple d'avoir à coder à l'intérieur des codes des interpolateurs qui comprennent le format des champs d'autres codes. L'API de couplage est aussi tout à fait compatible avec les types de couplages proposés par le lot R&D numérique [10].

Il a également été vérifié que les fonctionnalités offertes par les codes fins, les codes poreux et les codes système pourront être rendues accessibles via cette interface, et en respectant les spécifications. Cette vérification a porté sur Trio_U (et donc OVAP), FLICA [4], Cathare (et donc Neptune système) [5], ainsi que Neptune 3D local. Le cas des codes systèmes est le plus complexe, puisque ces codes sont pilotés par événements (ouverture de vanne,...), mais la notion de champ d'entrée est assez large pour pouvoir englober ces événements dans des champs très simplifiés.

Un soin particulier est apporté à la possiblité d'intégrer les couplages réalisés à partir de l'API de couplage dans le cadre du superviseur Salomé [13].

Enfin, le cas de couplage de Rupture de Tuyauterie Vapeur[5], utilisant à la fois Cathare, FLICA, et Chronos, semble également compatible avec l'interface proposée.

2.2 Les notions utilisées

L'API de couplage repose sur les notions de problème et de champ que nous allons définir ici. Pour autant, nous ne ferons ici aucun choix technique quant aux structures informatiques de ces objets.

Qu'est-ce qu'un champ ?

L'ensemble des informations échangées dans le cadre des couplages peut être représenté par des champs, c'est-à-dire des valeurs qui sont fonction des coordonnées d'espace *x*,*y*,*z* et du temps *t* sur un domaine limité de l'espace et du temps, appelé le support du champ. Les champs, ne pouvant être décrits que par un nombre fini de valeurs, doivent être discrétisés. Ils peuvent être par exemple décrits par des valeurs en des points précis ou par éléments, par des sommes pondérés de fonctions de formes, des coefficients de Fourier, des coefficients de polynômes, ...

On sépare donc l'information $f(x,y,z,t)$ en deux parties : d'une part les valeurs discrètes du champ, et d'autre part l'ensemble des informations qui permettent de reconstituer les valeurs $f(x,y,z,t)$ sur le support du champ. Ce sont notamment un domaine maillé et une discrétisation.

Pour les échanges de champs, l'API de couplage suppose l'existence d'une classe Field, sans en préciser l'implémentation.

Celle-ci est cependant plus riche que la classe correspondante de MED [14], qui pour le moment ne donne pas toute l'information sur la manière de reconstruire $f(x,y,z,t)$ à partir des valeurs discrètes. Elle pourra aussi inclure des notions géométriques complexes telles que les réseaux utilisés en neutronique. D'autre part, la classe Field doit inclure la notion de dépendance en temps. Par exemple, une implémentation simple de la dépendance en temps serait une collection de champs MED étiquetés par différents temps.

La définition précise de la classe Field fait partie des pré-requis au couplage. Dans le cas de couplages entre codes faisant partie de Neptune via Salomé, elles seront basées sur des extensions des classes MED actuelles. Mais dans le cas de couplages internes à un code, elles pourront être différentes de façon à se rapprocher des structures internes et optimiser le passage d'informations. C'est pourquoi la définition de la classe Field ne fait pas partie de ce document.

Qu'est-ce qu'un problème ?

On désigne par problème un objet informatique dont la fonction est de calculer un ou des champs inconnus sur un certain domaine d'espace et de temps. Le résultat peut être dépendant d'autres champs qui sont des paramètres d'entrée du problème. Les champs qui peuvent être fournis en sortie par le problème doivent être uniquement dépendants des inconnues. Par exemple, un problème peut prendre en entrée une vitesse transportante et un dépôt de puissance volumique et calculer une inconnue température. Les sorties seront alors des fonctions de la température. Un autre peut prendre en entrée des températures et calculer des flux neutroniques. Les sorties pourront être des taux de réactions ou des dépôts de puissance.

La nature informatique des problèmes n'est pas définie. Il peut s'agir d'un processus qui s'exécute sur une machine, d'un objet C++ ou python, d'un objet CORBA,... Ainsi, l'interface définie ici pourra être utilisée aussi bien pour des couplages entre codes qu'à l'intérieur d'un même code et d'un même processus. On ne définit pas la couche de communications, qui pourra être plus ou moins intelligente et orientée vers les interpolateurs parallèles comme MPCCI[8] ou FVM[9]. L'objet Problem est supposé déjà instancié et disponible : on ne traite pas ici de la façon dont on choisit de lancer tel code sur telle machine,... En particulier les méthodes de sauvegarde-reprise ne concernent qu'un même objet Problem, qui n'est pas détruit entre la sauvegarde et la reprise. Elles ne sont pas censées traiter de la reprise d'une simulation interrompue après destruction de l'objet Problem.

Coupler des problèmes revient à appeler dans un certain ordre différentes méthodes de l'API de couplage sur chacun des problèmes couplés. Les champs récupérés en sortie d'un problème peuvent être utilisés pour calculer les champs donnés en entrée d'un autre problème. Avec les méthodes proposées dans cette interface, on peut facilement créer des maquettes de couplage, qui, si l'interface est écrite en python pourront être très interactives. On peut aussi écrire des algorithmes de couplage génériques qui peuvent gérer l'avance en temps de n'importe quels problèmes présentant l'interface standard. Seuls les échanges de champs seront dépendants du cas de couplage envisagé.

Mais on peut aller plus loin : si le couplage est géré par une classe ou un code qui respecte cette même interface, alors le couplage lui-même sera un problème et pourra être couplé à d'autres problèmes. On pourra ainsi facilement créer des couplages de couplages, afin de résoudre par exemple à la fois plusieurs échelles et plusieurs disciplines.

Les exceptions

Les différentes méthodes de l'objet Problem sont prévues pour être utilisée d'une certaine façon, qui est spécifiée dans ce document. Dans le cas où elles seraient appelées de façon incorrecte, elles peuvent lever des exceptions. Il y a principalement deux types d'appels incorrects, qui vont provoquer des exceptions.

Le premier consiste à appeler une méthode sur un objet Problem alors qu'il est dans un état incompatible avec l'appel à cette méthode. C'est la chronologie des appels qui n'a pas été respectée. Dans ce cas, la méthode lève une exception de type `WrongContext`.

Le second consiste à appeler une méthode sur un objet Problem dont l'état interne le permet, mais les arguments passés à la méthode ne sont pas conformes aux spécifications. Dans ce cas la méthode lève une exception de type `WrongArguments`.

3 Définition de l'API de couplage

L'interface proposée est ici écrite en language IDL, parce que ce language est particulièrement explicite pour ce type de description.

Cependant, comme cette interface peut être utilisée à tous les niveaux, aussi bien dans Salomé pour coupler des codes qu'à l'intérieur d'un code pour coupler plusieurs problèmes, il n'est absolument pas indispensable de passer par CORBA, et elle peut légitimement être implémentée dans n'importe quel langage. Les solutions techniques d'implémentation ne sont pas imposées, seules les règles de comportement des méthodes sont définies. Mêmes les types des arguments peuvent différer légèrement d'une implémentation à l'autre. L'encapsulation ultérieure et les éventuelles conversions de types demanderont un travail de codage minime.

Cette interface est scindée en sept parties, qui regroupent les méthodes par thèmes : problèmes dépendants ou indépendants du temps, itératifs ou non, échange de champs, sauvegarde-reprise. L'arbre d'héritage est dessiné figure 1.

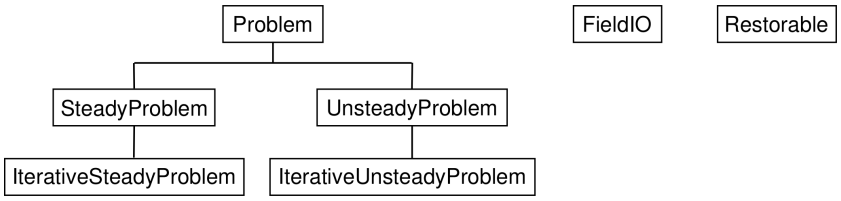


FIG. 1 – Arbre d'héritage des interfaces.

Règle 1 *Tout objet* `Problem` *doit au minimum implémenter soit l'interface* `SteadyProblem`*, soit l'interface* `UnsteadyProblem`*. Il peut implémenter des interfaces plus fines comme* `IterativeSteadyProblem` *ou* `IterativeUnsteadyProblem`*, et des interfaces complémentaires comme* `FieldIO` *et* `Restorable`*.*

```
interface Problem {
  initialize(in any input) raises WrongContext, WrongArguments;
  terminate() raises WrongContext;
};

interface SteadyProblem : Problem {
  boolean solve() raises WrongContext;
};

interface IterativeSteadyProblem : SteadyProblem {
  boolean iterate(out boolean converged) raises WrongContext;
};

interface UnsteadyProblem : Problem {
  double presentTime() raises WrongContext; // const
  boolean computeTimeStep(out double dt) raises WrongContext, WrongArguments; // const
  initTimeStep(in double dt) raises WrongContext, WrongArguments;
  boolean solveTimeStep() raises WrongContext;
  validateTimeStep() raises WrongContext;
  boolean isStationary() raises WrongContext; // const
  abortTimeStep() raises WrongContext;
};
```

```
interface IterativeUnsteadyProblem : UnsteadyProblem {
  boolean iterateTimeStep(out boolean converged) raises WrongContext;
};

interface FieldIO {
  sequence<string> getInputFieldsNames() raises WrongContext; // const
  Field getInputFieldTemplate(in string name) raises WrongContext, WrongArguments; // const
  setInputField(in string name,
                in Field afield) raises WrongContext, WrongArguments;
  sequence<string> getOutputFieldsNames() raises WrongContext; // const
  Field getOutputField(in string name) raises WrongContext, WrongArguments; // const
};

interface Restorable {
  save(in long id) raises WrongContext; // const
  restore(in long id) raises WrongContext, WrongArguments;
  forget(in long id) raises WrongContext, WrongArguments; // const
};
```

Ces interfaces ne sont pas réentrantes : une méthode doit impérativement être terminée avant qu'il soit possible d'en appeler une autre sur le même objet Problem. Les conditions dans lesquelles les méthodes lèvent l'exception `WrongContext` font l'objet de la section 4. Elles ne seront donc pas spécifiées dans la présente section.

3.1 interface `Problem`

Ici sont regroupées deux méthodes communes à tous les problèmes : celles de début et de fin de vie.

`initialize(in any input) raises WrongContext, WrongArguments;`

Cette méthode initialise le problème. Elle peut utiliser des informations présentes dans le paramètre qui est passé en argument. Elle effectue toutes les allocations et initialisations nécessaires aux appels des autres méthodes. La méthode `initialize` est appelée une fois et une seule avant toute autre méthode de l'interface.

argument input (lecture seule) Cet argument n'a pas de type défini, car il est très variable selon le code ou le problème considéré. Cela pourra être le nom d'un fichier d'entrée, des données Xdata, ou toute autre chose utile au problème.

exceptions

- `WrongContext`
- `WrongArguments` : si l'argument input n'est pas compris ou incorrect.

En cas d'erreur il faut terminer le calcul et appeler `terminate`.

droits Cette méthode peut modifier le problème.

`terminate() raises WrongContext;`

Cette méthode libère la mémoire utilisée et sauve ce qui doit l'être. Elle doit être appelée après toutes les autres en fin d'exécution du calcul ou en cas d'abandon du calcul. Après appel de `terminate`, plus aucune autre méthode ne peut être appelée.

exceptions
- `WrongContext`

droits Cette méthode peut modifier le problème.

3.2 interface `SteadyProblem`

Cette interface est utilisée pour tous les problèmes indépendants du temps. Dans ce cas, la méthode `initialize` a en charge l'initialisation des inconnues et des champs d'entrée sur $]-\infty,\infty[$. Ainsi, quelques que soient les méthodes d'échange de champs appelées même avant la résolution, les valeurs seront disponibles (même si elles sont arbitraires).

`boolean solve() raises WrongContext;`

Cette méthode fait l'ensemble de la résolution, c'est-à-dire qu'elle calcule les inconnues connaissant les champs d'entrée.

valeur de retour
- `true` : ok, les champs de sortie sont solution du problème.
- `false` : erreur dans la résolution, par exemple non convergence. Les champs de sortie ne sont pas solution du problème. Il peut être nécessaire de modifier les champs d'entrée pour obtenir la convergence.

exceptions
- `WrongContext`

droits Cette méthode peut modifier les inconnues du problème.

3.3 interface `IterativeSteadyProblem`

Lorsque la résolution d'un problème est effectuée de manière itérative, il peut être utile d'accéder séparément à chaque itération, par exemple pour modifier les champs d'entrée entre les itérations. C'est ce que permet cette interface, qui se compose d'une unique méthode en plus de l'interface `SteadyProblem`.

`boolean iterate(out boolean converged) raises WrongContext;`

Cette méthode effectue une itération unique, et met à jour l'inconnue en conséquence.

argument converged (sortie)
- `true` : la résolution est convergée, les champs de sortie sont solution du problème.
- `false` : la résolution n'est pas convergée, les champs de sortie ne sont pas solution du problème.

Notons que la boucle des itérations peut être arrêtée avant que `converged` soit mis à `true`, ou continuer même si `converged` vaut déjà `true`.

valeur de retour
- `true` : le problème est en cours de convergence (ou convergé).
- `false` : Le problème ne va pas converger. Les champs de sortie sont probablement très loin de la solution.

exceptions
- `WrongContext`

droits Cette méthode peut modifier les inconnues du problème.

Règle 2 *Un appel à* `solve` *doit être équivalent au pseudo-code suivant, qui doit s'exécuter en un temps fini :*

```
mettre converged à false
tant que converged est false
   si iterate(converged) est false
       renvoyer false
renvoyer true
```

Cela signifie qu'une suite d'appels à `iterate` doit toujours finir par renvoyer `false` ou `converged=true`, au bout d'un nombre fini d'appels. Cela traduit le fait qu'une suite est soit convergente soit divergente.

3.4 interface `UnsteadyProblem`

Un problème dépendant du temps possède des inconnues et des champs d'entrée qui sont tous dépendants du temps. A moins de conserver en mémoire l'ensemble des données sur toute la durée de la résolution, ce qui n'est pas généralement le cas, il doit effectuer sa résolution sur des intervalles de temps successifs, en oubliant progressivement les valeurs devenues inutiles.

Il faut donc porter une attention particulière à la définition des temps et des intervalles mis en jeu. Celle-ci est assez intuitive, de manière à ce que le comportement des problèmes soit aussi proche que possible de ce qu'on puisse attendre.

Le problème possède un temps présent t, accessible via la méthode `presentTime`. Toutes les valeurs de champs sur $]-\infty,t]$ sont non modifiables, elles appartiennent au passé. Elles peuvent aussi être oubliées par le problème dès qu'il ne les juge plus utiles.

Le problème possède aussi un intervalle de résolution $]t,t+dt]$, où dt la valeur du pas de temps. Celle-ci est imposée via la méthode `initTimeStep`. Une valeur suggérée par le problème peut être obtenue via la méthode `computeTimeStep`. C'est sur cet intervalle que se font les échanges de champs, conformément à la règle 7. C'est aussi sur cet intervalle que les méthodes de résolution vont calculer les inconnues en fonction des champs d'entrée, et des valeurs passées.

Ainsi la résolution d'un pas de temps se fait en trois étapes. Tout d'abord, `initTimeStep` alloue et initialise les champs sur l'intervalle de résolution. A partir de là ils sont disponibles aux méthodes d'échange de champs. Puis le calcul des inconnues sur l'intervalle de résolution est fait par des appels à `solveTimeStep` ou `iterateTimeStep`. Si le résultat est satisfaisant, il est validé et le présent est avancé à $t+dt$ par `validateTimeStep`. Sinon, le calcul sur l'intervalle de temps est abandonné par appel à `abortTimeStep`. Dans les deux cas, l'intervalle devient inaccessible de l'extérieur tant en lecture qu'en écriture.

La méthode `initialize` a en charge l'initialisation des inconnues et des champs d'entrée à l'instant initial.

`double presentTime() raises WrongContext;`

Cette méthode permet d'obtenir le temps présent t. Elle peut être appelée n'importe quand après `initialize`.

valeur de retour Le temps présent renvoyé par le problème.

exceptions
- `WrongContext`

droits Méthode const, ne peut pas modifier le problème.

Règle 3 *La valeur du temps présent telle que renvoyée par* `presentTime` *ne peut changer que lors de l'appel à* `validateTimeStep`*.*

`boolean computeTimeStep(out double dt) raises WrongContext;`

Cette méthode permet d'obtenir la valeur du pas de temps *dt* souhaitée par le problème. Ce pas de temps se réfère toujours à un intervalle de résolution commençant au temps présent *t* du problème. La valeur retournée n'est qu'indicative. La méthode `computeTimeStep` peut être appelée n'importe quand après `initialize`.

argument dt (sortie) Le pas de temps souhaité renvoyé par le problème.

valeur de retour
- `true` : le problème souhaite continuer avec le pas de temps rendu dans `dt`.
- `false` : le problème souhaite s'arrêter. Alors la valeur de `dt` n'a pas de sens.

exceptions
- `WrongContext`

droits Méthode const, ne peut pas modifier le problème. Elle peut utiliser les inconnues et les champs d'entrée sur les domaines où ils sont définis.

`initTimeStep(in double dt) raises WrongContext;`

Cette méthode alloue et initialise les inconnues et les champs d'entrée sur l'intervalle $]t, t+dt]$, par exemple à partir des valeurs à des temps antérieurs.

Les initialisations doivent être faites avec soin. En effet, après cette méthode, les champs de sortie sont disponibles, donc les inconnues peuvent être utilisées via les appels à `getOutputField`, et les champs d'entrée peuvent être utilisés dans la résolution s'ils ne sont pas écrasés par des appels à `setInputField`. Le calcul utilisé pour les initialisations doit être explicité dans la documentation du problème. Dans le cas le plus simple, cela revient simplement à recopier les valeurs présentes des champs dans les valeurs futures.

argument dt (lecture seule) La valeur du pas de temps sur lequel il faut allouer et initialiser les champs.

exceptions
- `WrongContext`

droits Cette méthode peut modifier le problème.

`boolean solveTimeStep() raises WrongContext;`

Cette méthode est analogue à la méthode `solve` du cas indépendant du temps : elle calcule les inconnues sur l'intervalle $]t, t+dt]$, connaissant les champs d'entrée et les valeurs passées.

valeur de retour
- `true` : ok, les champs de sortie sont solution sur $]t,t+dt]$.
- `false` : erreur dans la résolution, par exemple non convergence. Les champs de sortie ne sont pas solution du problème. Il peut être nécessaire de modifier les champs d'entrée ou la valeur de dt pour obtenir la convergence.

exceptions
- `WrongContext`
- `WrongArguments` : dt est négatif ou nul.

droits Cette méthode peut modifier les inconnues du problème sur $]t,t+dt]$.

`validateTimeStep() raises WrongContext;`

Cette méthode valide le calcul effectué sur l'intervalle de résolution en avançant la valeur du présent à $t+dt$. De ce fait les champs ne peuvent plus être modifiés sur l'ancien intervalle de résolution, et ils ne sont plus accessibles, jusqu'au prochain appel à `initTimeStep` qui définit un nouvel intervalle de résolution. Le problème peut profiter de l'appel à `validateTimeStep` pour libérer des valeurs passées des champs.

exceptions
- `WrongContext`

droits Cette méthode peut modifier la valeur du temps présent.

`boolean isStationary() raises WrongContext;`

Cette méthode ne peut être appelée qu'après le calcul sur un pas de temps, c'est-à-dire après appel à `validateTimeStep`.

Elle indique si le problème est resté stationnaire ou non lors du dernier pas de temps. Cette valeur est indicative : il reste possible de continuer la résolution même après avoir atteint le stationnaire.

valeur de retour
- `true` : l'état stationnaire est atteint
- `false` : l'état stationnaire n'est pas atteint.

exceptions
- `WrongContext`

droits Méthode const, ne peut pas modifier le problème.

`abortTimeStep() raises WrongContext;`

Cette méthode abandonne le calcul effectué sur l'intervalle de résolution $]t,t+dt]$. Elle est appelée à la place de `validateTimeStep`, par exemple si la résolution a échoué ou si l'on souhaite modifier la valeur de dt. Elle libère les champs de l'intervalle $]t,t+dt]$, qui deviennent donc inaccessibles. Elle laisse inchangé le temps t.

exceptions
- `WrongContext`

droits Cette méthode peut modifier le problème.

3.5 interface `IterativeUnsteadyProblem`

Comme `IterativeSteadyProblem` introduit la notion d'intérations dans la résolution de `SteadyProblem`, `IterativeUnsteadyProblem` introduit la notion d'itérations dans la résolutin d'un pas de temps de `UnsteadyProblem`.

`boolean iterateTimeStep(out boolean converged) raises WrongContext;`

Cette méthode est analogue à la méthode `iterate` des problèmes indépendants du temps : elle effectue donc une itération unique de la résolution sur l'intervalle $]t,t+dt]$, et met à jour l'inconnue en conséquence.

argument converged (sortie)
- `true` : la résolution est convergée, les champs de sortie sont solution du problème sur $]t,t+dt]$.
- `false` : la résolution n'est pas convergée, les champs de sortie ne sont pas solution du problème sur $]t,t+dt]$.

La boucle des itérations peut être arrêtée avant ou après que le problème s'estime convergé.

valeur de retour
- `true` : le problème est en cours de convergence (ou convergé) sur $]t,t+dt]$.
- `false` : Le problème ne va pas converger. les champs de sortie sont probablement très loin de la solution sur $]t,t+dt]$.

exceptions
- `WrongContext`

droits Cette méthode peut modifier les inconnues du problème sur $]t,t+dt]$.

Règle 4 *(analogue à la règle 2) Un appel à* `solveTimeStep` *doit être équivalent au pseudo-code suivant, qui doit s'exécuter en un temps fini :*

```
mettre converged à false
tant que converged est false
   si iterateTimeStep(converged) est false
       renvoyer false
renvoyer true
```

Cela signifie qu'une suite d'appels à `iterateTimeStep` doit toujours finir par renvoyer `false` ou `converged=true`, au bout d'un nombre fini d'appels.

3.6 interface `FieldIO`

Tout problème qui vent échanger des champs avec l'extérieur doit le faire via l'interface `FieldIO`. Une grande attention doit être portée à la spécification des champs échangés, pour que le couplage soit cohérent.

3.6.1 Séparation des entrées et des sorties

Tout d'abord, il convient de définir les champs d'entrée et de sortie. L'objet Problem calcule des inconnues en fonction de paramètres d'entrée. Les champs que le problème peut donner en sortie doivent dépendre exclusivement

des inconnues qu'il calcule. Les champs qu'il prend en entrée sont des paramètres de calcul, et ne peuvent en aucun cas écraser les inconnues. Chaque champ d'entrée ou de sortie est repéré par son nom, qui est une chaîne de caractères. Séparer ainsi les entrées et les sorties permet de satisfaire la règle suivante (très importante) :

Règle 5 *Les appels aux méthodes de l'interface* `FieldIO` *doivent être commutatifs entre eux.*

On appelle "bloc d'échange de champs" (pour un objet Problem donné) une suite d'appels consécutifs à des méthodes de l'interface `FieldIO`. La règle 5 signifie que l'ordre chronologique des appels à ces méthodes à l'intérieur d'un même bloc d'échange de champs n'a pas d'influence sur les résultats. Notamment, le fait d'imposer un champ d'entrée ne doit pas affecter la valeur rendue par un champ de sortie.

3.6.2 Extension temporelle des champs échangés

Comme un champ représente une ou des valeurs fonction de l'espace et du temps sur un certain domaine de définition, il convient de préciser ce domaine. Le domaine spatial, ainsi que la discrétisation, sont à la discrétion du problème. Dans le cas de champs en sortie, ils sont disponibles en lecture puisque le champ porte la définition de son support. Dans le cas de champs en entrée, ils sont disponibles via la méthode `getInputFieldTemplate`.

Comme on l'a vu en section 3.4, le temps dispose d'un statut particulier, lié au fait que le passé ne peut être modifié, et que le déroulement de la simulation correspond à des temps croissants continument (sauf appels aux méthodes de sauvegarde-reprise). De ce fait, l'extension temporelle des champs échangés est imposée par les deux règles suivantes.

Règle 6 *Dans le cas d'un problème indépendant du temps, les champs sont par convention définis sur* $]-\infty,\infty[$ *et constants dans le temps.*

Règle 7 *Dans le cas d'un problème dépendant du temps, les champs échangés sont tous définis sur l'intervalle* $]t,t+dt]$ *où* t *est l'instant présent tel que renvoyé par* `presentTime`*, et* dt *la valeur du pas de temps telle qu'imposée lors du dernier appel à* `initTimeStep`*.*

Le déroulement chronologique tel que défini en section 4 assure que cet intervalle est bien défini à l'intérieur d'un bloc d'échange de champs.

`sequence<string> getInputFieldsNames() raises WrongContext;`

Cette méthode renvoie la liste des noms des champs d'entrée du problème. Sa présence facilite l'automatisation des couplages, sans connaissance préalable des problèmes couplés.

valeur de retour C'est la liste des noms des champs d'entrée du problème. La destruction de cette liste est à la charge de l'appelant. Tout appel à `setInputField` dans le même bloc d'échange de champs doit utiliser un des noms de cette liste.

exceptions

- `WrongContext`

droits Méthode const, ne peut pas modifier le problème.

`Field getInputFieldTemplate(in string name) raises WrongContext, WrongArguments;`

Cette méthode permet d'obtenir la forme sous laquelle le problème attend le champ d'entrée de nom `name`. La sortie est un champ dont les valeurs discrètes sont absentes, ou n'ont pas de signification. Seules les autres

informations sont pertinentes, notamment le support et la discrétisation. Dans le cas d'un problème dépendant du temps, ce champ a obligatoirement pour extension temporelle l'intervalle $]t, t+dt]$, et sinon il doit être constant sur $]-\infty, \infty[$.

argument name (lecture seule) Le nom du champ d'entrée dont on souhaite obtenir le modèle.

valeur de retour Le modèle sur lequel le problème attend le champ d'entrée de nom `name`. Ce champ doit rester indéfiniment valide et sa destruction est à la charge de l'appelant. Tout appel à `setInputField` dans le même bloc d'échange de champs, et pour un champ de même nom, doit fournir un champ sur ce modèle.

exceptions
- `WrongContext`
- `WrongArguments` : le nom `name` ne fait pas partie des noms rendus par `getInputFieldsNames`.

droits Méthode const, ne peut pas modifier le problème.

`setInputField(in string name, in Field afield) raises WrongContext, WrongArguments;`

argument name (lecture seule) Le nom du champ d'entrée que l'on souhaite imposer.

argument afield (lecture seule) Le champ d'entrée que l'on souhaite imposer. Ce champ doit pouvoir être détruit par l'appelant dès la sortie de la méthode. A noter : l'appel à `setInputField`, comme à toutes les autres méthodes d'échange de champ, est toujours facultatif, puisque les champs d'entrée sont initialisés lors de `initialize` et de `initTimeStep`.

exceptions
- `WrongContext`
- `WrongArguments` : le nom `name` ne fait pas partie des noms rendus par `getInputFieldsNames`, ou le champ `afield` n'est pas conforme au modèle rendu par `getInputFieldTemplate(name)`.

droits Le problème peut être modifié, mais seulement dans la limite de la règle 5, c'est-à-dire en pratique que `setInputField` ne modifie pas les inconnues du problème.

`sequence<string> getOutputFieldsNames() raises WrongContext;`

Cette méthode renvoie une liste de noms de champs de sortie du problème. Sa présence facilite l'automatisation des couplages, sans connaissance préalable des problèmes couplés.

Les noms rendus en valeur de retour doivent être des noms de champs de sortie valables tout au long de l'exécution du problème, c'est-à-dire que tout appel à `getOutputField` avec un nom de cette liste doit aboutir. En revanche, cette liste n'est pas exhaustive, et d'autres noms de champs de sorties peuvent être valables (en particulier rien n'empêche qu'il y en ait un nombre infini...)

valeur de retour Une liste de noms de champs de sorties valables, qui pourront être récupérés via `getOutputField`. La destruction de la liste est à la charge de l'appelant.

exceptions
- `WrongContext`

droits Méthode const, ne peut pas modifier le problème.

`Field getOutputField(in string name) raises WrongContext, WrongArguments;`

Cette méthode renvoie un champ de sortie du problème, correspondant au nom passé dans l'argument `name`.

argument name (lecture seule) Le nom du champ de sortie que l'on souhaite obtenir.

valeur de retour Le champ de nom `name`, rendu par le problème. Ce champ doit rester indéfiniment valide et sa destruction est à la charge de l'appelant.

exceptions
- `WrongContext`
- `WrongArguments` : le nom `name` n'est pas reconnu comme champ de sortie. Notons que si `name` fait partie de la liste rendue par `getOutputFieldsNames`, il doit être reconnu.

droits Méthode const, ne peut pas modifier le problème. Afin de vérifier la règle 5, le champ rendu ne doit dépendre que des inconnues du problème.

3.7 interface `Restorable`

Cette interface regroupe es trois méthodes qui permettent à un problème de retrouver l'état qu'il avait à un moment antérieur de son exécution.

`save(in long id) raises WrongContext;`

Cette méthode sauvegarde l'état interne du problème en lui associant un identificateur. La forme de la sauvegarde est laissée libre (support, format, ...).

Si `save` a déjà été appelé avec le même argument `id`, la précédente sauvegarde est écrasée par l'état actuel.

argument id (lecture seule) L'identificateur à associer à l'état sauvegardé.

exceptions
- `WrongContext`

droits Méthode const, ne peut pas modifier le problème

`restore(in long id) raises WrongContext, WrongArguments;`

Cette méthode restaure un état précédemment sauvegardé via la méthode `save` par le même objet Problem, et sous le même identificateur.

argument id (lecture seule) L'identificateur associé à l'état qui doit être restauré.

exceptions
- `WrongContext`
- `WrongArguments` : aucun état n'a été sauvegardé sous l'identificateur `id`, ou la sauvegarde est corrompue.

droits Cette méthode peut modifier le problème

Règle 8 *Après un appel réussi à* `restore`*, les appels à d'autres méthodes doivent donner exactement les mêmes résultats (à la précision machine près) que s'ils avaient été effectués après le* `save` *correspondant.*

Sont exclus de cette règle les appels aux méthodes de sauvegarde et reprise, la liste des états sauvegardés ayant pu changer entre les appels à `save` et `restore`.

`forget(in long id) raises WrongContext, WrongArguments;`

Cette méthode libère ou "oublie" un état sauvegardé par la méthode `save` sous le même identificateur. Elle sert à libérer de la place disque ou mémoire, lorsque l'état sauvegardé est devenu inutile.

argument id (lecture seule) L'identificateur associé à l'état qui doit être libéré.

exceptions
- `WrongContext`
- `WrongArguments` : aucun état n'a été sauvegardé sous l'identificateur `id`, ou la sauvegarde est corrompue.

droits Méthode const, ne peut pas modifier le problème

4 Chronologie des appels

A la lecture de l'API de couplage, il apparaît clairement que les méthodes ne peuvent pas être appelées dans n'importe quel ordre. En particulier, il faut qu'une variable soit définie avant de pouvoir être utilisée. .

Une partie des conditions a déjà été explicitée dans la définition des méthodes, mais la façon la plus synthétique de définir à quel moment il est licite d'appeler une méthode ou non, est de fixer l'ordre chronologique des appels, et de le représenter sous forme de diagramme d'exécution. Cette section présente deux diagrammes d'exécution commentés, l'un dans le cas des problèmes indépendants du temps et l'autre dans le cas des problèmes dépendants du temps.

Si la chronologie n'est pas respectée et qu'une méthode est appelée hors de ces diagrammes, l'état du problème ne permet pas l'exécution de la méthode : elle est alors censée lever l'exception `WrongContext`.

4.1 Comment lire les diagrammes

Pour des raisons de lisibilité, certaines méthodes sont absentes des diagrammes d'exécution.

Les méthodes de l'interface `Restorable`, ainsi que dans le cas dépendant du temps les méthodes `presentTime` et `computeTimeStep`, peuvent être appelées n'importe quand après `initialize` et avant `terminate`. A part la méthode `restore`, elles ne modifient pas le problème, donc elles n'ont aucun impact sur le déroulement du diagramme d'exécution. La méthode `restore` a pour effet de replacer le problème dans l'état où il était lors de l'appel à la méthode `save` correspondant, et donc au même emplacement dans le diagramme d'exécution.

Deux diagrammes sont sont présentés, selon que le problème implémente l'interface `SteadyProblem` (et d'autres éventuellement) ou l'interface `UnsteadyProblem` (et d'autres éventuellement).

4.2 Problèmes indépendants du temps

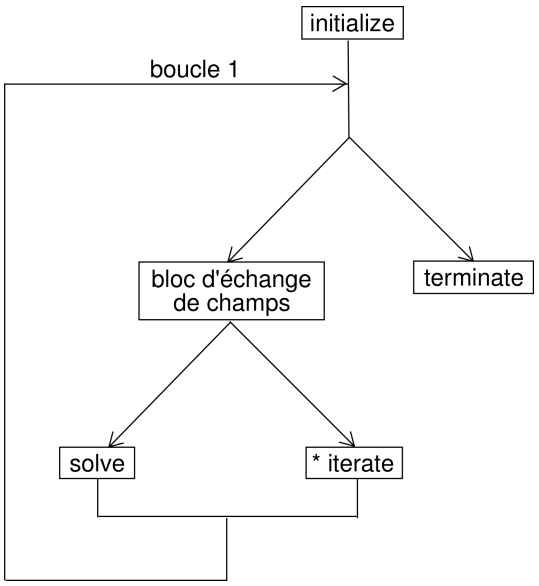


FIG. 2 – Diagramme d'exécution d'un problème indépendant du temps. Le signe étoile signifie : "un nombre quelconque d'appels à", ce nombre pouvant être nul. Le bloc d'échange de champs correspond à un nombre quelconque d'appels aux méthodes de l'interface `FieldIO`, et ce dans un ordre quelconque.

Le diagramme d'exécution est représenté sur la figure 2.

Après l'initialisation du problème, les champs sont définis et les échanges de champs possibles. En particulier on pourra imposer les champs d'entrée. Puis on peut appeler les méthodes de résolution `solve` ou un nombre quelconque de fois `iterate`. Si la résolution est terminée, on peut appeler `terminate`. Notons que la résolution est facultative : on peut n'appeler ni `solve` ni `iterate`.

La boucle 1 permet de revenir au niveau des échanges de champs. On peut ainsi modifier les champs d'entrée et de relancer des étapes de résolution avec les nouvelles entrées. Par exemple, on peut modifier les champs d'entrée entre deux itérations du processus de résolution. Cette boucle permet également de lire les champs de sortie avant l'appel à `terminate`, voire entre chaque itération si cela est souhaitable. Le rôle de la boucle 1 est donc de permettre le couplage au niveau le plus fin possible.

4.3 Problèmes dépendants du temps

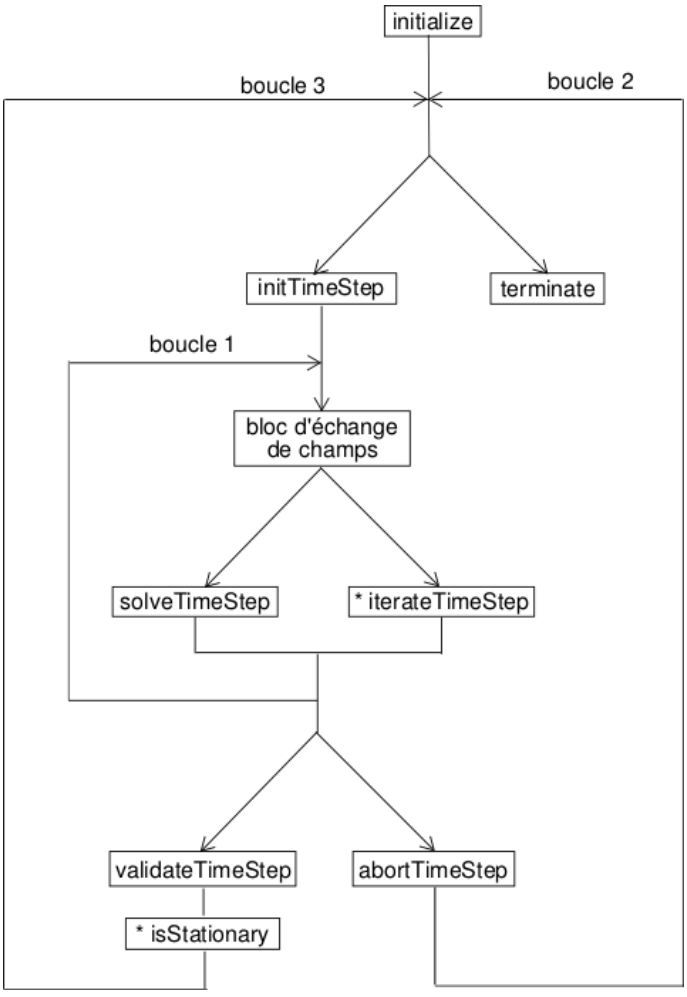


FIG. 3 – Diagramme d'exécution d'un problème dépendant du temps. Le signe étoile signifie : "un nombre quelconque d'appels à", ce nombre pouvant être nul. Le bloc d'échange de champs correspond à un nombre quelconque d'appels aux méthodes d'échange de champs, et ce dans un ordre quelconque.

Le diagramme d'exécution est représenté sur la figure 3.

Le coeur du diagramme est identique à celui des problèmes indépendants du temps. Pour la résolution d'un pas de temps, les mêmes possibilités de couplage sont offertes que pour la résolution d'un problème indépendant du

temps. Cependant, deux boucles supplémentaires sont nécessaires pour la gestion des pas de temps et l'avance en temps.

La résolution d'un pas de temps se fait lorsque l'intervalle $]t, t+dt]$ est défini et que les champs sont alloués et initialisés sur cet intervalle. C'est pourquoi elle est précédée d'un appel à `initTimeStep` qui est chargé de ces allocations et suivie d'un appel à `validateTimeStep` qui rend cet intervalle non modifiable en avançant le temps présent, ou bien à `abortTimeStep` qui le libère.

La boucle 2 permet de recommencer une nouvelle résolution de pas de temps sur un intervalle plus grand ou plus petit, en ayant abandonné le précédent. Ainsi, si par exemple le pas de temps imposé s'avère trop élevé et que la résolution ne converge pas, il est possible de recommencer avec un pas de temps plus petit. De nombreuses méthodes implicites ont besoin de cette possibilité.

La boucle 3 correspond à l'avance en temps. Une fois le présent avancé à la fin de l'intervalle de résolution, on peut allouer un nouvel intervalle et lancer la résolution dessus.

Enfin, la méthode `isStationary` peut être appelée une fois que le pas de temps est validé (par `validateTimeStep`).

5 Conclusion

L'interface présentée ici permet potentiellement de reproduire l'ensemble des couplages étudiés, dans un cadre uniforme où il sera plus facile de remplacer un code par un autre, et de minimiser les investissements nécessaires à la création d'un nouveau couplage, ou à l'utilisation d'un nouveau code.

Néanmoins, la définition de l'interface n'est qu'un premier pas dans la direction de couplages simples et généraux tels que décrits dans l'introduction. La prochaine étape pour le couplage entre codes Neptune consiste dans la définition précise de la classe Field à échanger. Elle sera bien entendu basée sur les classes MED, ou une extension de ces classes. Elles devra notamment intégrer la notion de dépendance en temps, et des informations pour la reconstruction du champ sur l'ensemble de son support. Puis il faudra implémenter cette interface dans différents codes, qui dès lors pourront être couplés de façon générique. Plus le nombre de codes qui adopteront l'interface sera important, plus le gain apporté par celle-ci sera substantiel. Dès à présent, l'adoption de la présente interface est envisagée pour le code NEPTUNE Système[16]. Enfin, il faudra écrire des coupleurs génériques en python puis dans le superviseur Salome. Une bibliothèque d'interpolateurs sera développée puis enrichie au cas par cas, en fonction des besoins.

Par ailleurs, il faudra aussi envisager des extensions de cette interface, qui présente encore quelques limitations et ne résoud pas tous les problèmes. Tout d'abord, on peut imaginer qu'un maillage dépende du résultat du problème auquel il appartient ou même d'un autre problème. Dans ce cas, il faut pouvoir imposer un maillage depuis l'extérieur. Le pas de temps lui-même peut dépendre des champs d'entrée et du maillage. Une autre limitation concerne les couplages 100% implicites, qui nécessitent le transfert de jacobiens que l'API de couplage ne permet pas encore.

Même sans rajouter de fonctionnalités supplémentaires, il peut être souhaitable d'étendre l'interface pour en rendre l'utilisation plus commode ou plus efficace. Ainsi on pourra penser à ajouter des méthodes "de confort" telles que l'échange d'une liste de champs en un seul appel, ou la possibilité d'avancer jusqu'à un temps donné, quelque soit le nombre de pas de temps utilisés. Ces méthodes correspondent à une suite d'appels à des méthodes décrites ici. On pourra également ajouter des méthodes "d'optimisation" qui permettront, dans certains cas, de réduire la quantité de données à échanger. On peut penser par exemple à passer un champ uniquement sur une partie du domaine définie par un masque, ou à indiquer que certaines propriétés sont invariables comme des supports, les listes de champs d'entrée/sortie,... Les méthodes de confort et d'optimisation ne sont volontairement pas incluses dans cette première version de l'interface pour la rendre plus lisible et plus compréhensible.

Lorsque de nouvelles versions de l'API de couplage seront publiées, il conviendra de veiller à ce qu'elles restent toujours compatibles avec les précédentes : tout objet implémentant l'API version 2 pourra être utilisé via l'API version 1, même si certaines fonctionnalités ne sont pas accessibles.

Mais une des questions les plus urgentes à résoudre sera sans doute celle du parallélisme : comment coupler entre eux des codes qui s'exécutent chacun en parallèle et possèdent des champs distribués ? La réponse immédiate est de faire repasser le champ sur un unique processeur pour les communications avec le coupleur et que les interpolations soient elles aussi monoprocesseur. Mais dans le cas de l'échange de champs 3D très volumineux, cette solution devient prohibitive, et il faudrait que les champs échangés soient eux-mêmes distribués, et que les interpolations se fassent en parallèle. Cela n'impose pas forcément de changer l'API de couplage, en revanche cela impose des contraintes très importantes sur la structure Field, ainsi que sur les coupleurs, les interpolateurs et le superviseur Salomé lui-même.

Références

[1] Description de l'architecture générale Neptune, Nept_2002_L1.1/9

[2] Composant Neptune Problem, compte-rendu de réunion du 19/03/2003, Nept_2003_L1.1/9

[3] Neptune component interpolation reference manual and user's guide, NT/DTN/SMTM/2004-41

[4] FLICA4 : encapsulation Python / SALOME, Nept_2004_L1.3/2

[5] NEPTUNE : Cas Test Rupture de Tuyauterie Vapeur - Mise en oeuvre et résultats, Nept_2004_L1.3/3

[6] ISAS : système de couplage de codes. Présentation de Alain Bengaouer

[7] Calcium : l'outil de couplage de codes de EDF-R&D. Présentation de Jean-Yves Berthou

[8] MPCCI : multidisciplinary simulations through code coupling, http ://www.scai.fraunhofer.de/mpcci.html

[9] Point location and variable exchange with the FVM library : basic principles and distributed mode, Présentation de Yavan Fournier.

[10] Couplage de deux systèmes de la dynamique des gaz, Nept_2005_L2.1/11

[11] Les couplages dans Neptune : réflexions sur l'architecture des applications, Nept_2005_L1.1/5

[12] Test d'interoprabilité Choc Froid Diphasique, Nept_2004_L1.4/1

[13] http ://www.salome-platform.org

[14] MED Module Documentation, http ://www.salome-platform.org/ex/doc/MED/MED_index.html

[15] Différentes approches pour couplage Trio_U-Trio_U et Trio_U-Cathare, STH/LMDL_2005_043

[16] Spécifications de l'application Neptune-Système version V1, Nept_2005_L1.4/3

[17] Validating and Verifying a new thermal-hydraulic analysis tool

[18] The TE coupled RELAP5/PANTHER/COBRA code package and methodology for integrated PWR accident analysis