N° d'ordre : 2533

Thèse présentée pour obtenir

Le titre de docteur de l'Institut National Polytechnique de Toulouse

École doctorale : Systèmes

Spécialité : Systèmes industriels

Par M **Karim ALLOULA**

Ingénieur E.N.S.E.E.I.H.T.

Titre de la thèse:

Modèle de coopération entre calcul formel et calcul numérique pour la simulation et l'optimisation des systèmes

Soutenue le 9 novembre 2007 devant le jury composé de :

Pr.	Xavier JOULIA	Président
Pr.	Jean-Marc LE LANN	Directeur de thèse
Dr.	Bertrand BRAUNSCHWEIG	Rapporteur
Pr.	Sandro MACCHIETTO	Rapporteur
M.	Olivier BAUDOUIN	Membre
Dr.	Jean-Pierre BELAUD	Membre

Résumé

Modèle de coopération entre calcul formel et calcul numérique pour la simulation et l'optimisation des systèmes.

Après avoir étudié les collaborations établies aujourd'hui entre différents environnements de résolution de problèmes, le manuscrit propose un modèle de conception d'un système de calcul basé sur la coopération entre calcul formel et numérique. Cette coopération entre différents sous-systèmes de calcul est de type complémentaire : les rôles sont définis a priori.

Suivant une démarche orientée modèle, le modèle de coopération est spécifié en UML 2.0 selon la vue structurelle et la vue comportementale. A partir du modèle conceptuel, nous définissons les règles de transformation pour produire le modèle d'implémentation spécifique de la « plate-forme » FORTRAN 90.

Au vu des résultats d'études particulières en génie des procédés -la solvatation d'acides forts, et la distillation de Rayleigh- il apparaît que la démarche de calcul coopératif proposée :

- apporte plus d'expressivité lors de la modélisation;
- incite à modéliser les systèmes physiques à l'aide de fonctions, souvent implicites ;
- permet la réutilisation de modèles par la composition et l'assemblage de fonctions ;
- et apporte plus de fiabilité lors de la simulation, notamment grâce au calcul précis des dérivées des modèles.

Mots-clés : Calcul Formel, Calcul Numérique, Modèle de Coopération, Systèmes Industriels, Génie des Procédés.

Abstract

A co-operative model combining computer algebra and numerical calculation for system simulation and optimization.

After investigating state-of-the-art collaborations between various environments aimed at system resolution, this paper presents a design model for a calculation system based on the co-operation of formal and numerical calculations. This co-operation between multiple sub-systems is complementary: the roles are defined a priori.

Following an object-oriented approach, the model is specified via UML 2.0, in terms of the structural and behavioural views. From the conceptual model, we define the transformation rules required to create the implementation-specific model for the FORTRAN 90 platform.

From the results witnessed within specific process engineering studies - namely the solvation of strong acids and Rayleigh's distillation - it can be seen that this co-operative approach:

- empowers us with an improved expressivity at the modeling stage;
- instigates physical modeling using (often implicit) functions;
- allows model re-use through function aggregation and assembly;
- and brings a greater reliability during simulation, notably as a result of the precise calculation of derivatives of the model.

Keywords: Computer Algebra, Numerical Calculation, Co-operation Model, Industrial Systems, Process Engineering.

A Valérie et Louise.

Remerciements

Merci à mes anciens collègues d'Elf Aquitaine qui au tout début de ma carrière professionnelle m'ont démontré qu'il était possible d'allier plaisir scientifique et utilité industrielle. Au premier rang de ces habiles numériciens je place Dominique Apprato et Claude Leibovici qui ont toute mon amitié. Je n'oublie ni Jean-Luc Boëlle, ni Peppino Terpollili, ni Marc Verprat, ni Bernard Pflugfelder, ni Pascal Klein, ni Guy Canadas, ni Guy Barré, ni Jacky Bernier, ni Alain Tomasian, ni Yogeshwar Sharma, ni Olivier Gosselin, ni François Chapel.

Merci à mes anciens collègues de Vérilog qui eux m'ont fait goûter aux joies de développer du logiciel dans un contexte formalisé, soucieux de la qualité. Sayed Accari, Manuel Barroso et Elena Ivanova ont partagé mon bureau. C'est grâce à eux notamment que je suis sensible aux notions de modèle de conception, de modèle d'implémentation et autres apports du génie logiciel que j'ai essayé d'employer dans le travail présenté… Hervé Leblanc, maître de conférences et chercheur à l'Institut de Recherche en Informatique de Toulouse (I.R.I.T.) a su rafraîchir ces notions acquises il y a déjà longtemps en m'ouvrant au domaine de l'ingénierie logicielle dirigée par les modèles.

Merci à tous mes collègues actuels de l'Institut National Polytechnique de Toulouse, qui chacun dans son domaine me montrent au quotidien que « conscience professionnelle » n'est pas un vain mot. Jean-Pierre Belaud, Cédric Brandam, Stéphane Négny et Christian Hervé partagent mes repas de midi, souvent un café… Ils ont donc porté un regard des plus bienveillants sur mon activité de recherche ! Béatrice Guinard, Cyril Cotonat, Guilhem Petit et Michel Richard partagent les heures où je joue le rôle d'ingénieur informaticien. Ils m'ont donc connu soucieux durant les mois de rédaction et m'ont épaulé lorsque j'avais du mal à endosser mes deux rôles…

Merci à Jean-Claude Yakoubsohn du laboratoire de Mathématiques pour l'Industrie et la Physique (M.I.P.), qui a eu une oreille attentive lorsque je lui exposais mes espoirs de numéricien. Les acquis de son équipe dans le domaine de la résolution des systèmes non linéaires m'ont permis de tirer la substantifique moelle de la méthode de Newton. Je suis impatient de faire de même pour les systèmes d'équations algébro-différentielles sur lesquels nous commençons à travailler ensemble.

Merci à Vincent Coll dont le remarquable travail d'ingénieur a permis que eXMSL soit un petit peu plus que le résultat habituel d'un développement logiciel dans le cadre d'une thèse. L'interface graphique produite autour des services de calcul numérique et formel les valorise en les rendant accessibles aisément et à distance.

Merci aux collègues devenus des amis ! Françoise Bellon, Jean Martinez, Hervé Pingaud, Philippe Duquenne, Joon Suh Lee, Mike Chaplee et Knut Bredahl sont de ceux-là.

Merci enfin à tous les membres du jury qui ont porté la plus grande attention scientifique à ce travail, et ont enrichi le débat par leurs remarques. Ils connaissent toute la considération que j'ai pour chacun d'eux, et je ne suis pas le seul ! Ayant partagé un bout de chemin, voire un long chemin avec chacun d'eux, cette considération est non seulement professionnelle mais aussi personnelle.

Toutes les personnes précitées, et évidemment ma famille, m'ont incité à accomplir ce travail de recherche ou m'y ont aidé.

Papier, crayon…

La qualité des environnements de calcul formel proposés aujourd'hui sur le marché permet parfois de concilier les objectifs de fiabilité et de performance visés par l'ingénieur et le chercheur chargés de modéliser, simuler ou optimiser un système. Certaines simulations peuvent tout à fait prendre place dans le cadre de ces outils uniquement. La plupart toutefois nécessitent des ressources spécifiques, quelles soient logicielles, matérielles, ou le plus souvent dictées par le domaine d'application. Elles peuvent difficilement avoir lieu dans un environnement de calcul formel livré clé en main, et prennent plutôt place dans un environnement de simulation numérique, souvent constitué comme une chaîne d'outils logiciels. Une réponse satisfaisante n'est-elle pas alors de faire coopérer les environnements de simulation numérique avec des environnements de calcul formel, les modèles mathématiques étant au cœur de cette coopération entre systèmes logiciels ? Dès lors, les expressions mathématiques constituant les modèles seront l'objet alternativement d'étapes de transformation formelle et d'étapes d'évaluation numérique. Chaque étape exploitera au mieux les qualités de chacun des systèmes de calcul utilisés : la performance pour le système de calcul numérique, la fiabilité pour le système de calcul formel. Ce modèle de coopération entre calcul formel et calcul numérique reproduit en fait ce qui se passe, et ce de façon transparente, lorsque nous avons à mener des calculs avec un crayon et du papier : nous alternons des étapes de manipulation algébrique des expressions et des étapes de calcul numérique.

TABLE DES MATIERES

FIGURES

EQUATIONS

TABLEAUX

Introduction

Le but de ce mémoire est de proposer un modèle original de coopération entre le calcul formel et le calcul numérique dans le but d'améliorer les processus de modélisation, de simulation et d'optimisation des systèmes statiques et dynamiques.

Si les résultats obtenus aujourd'hui par la modélisation et la simulation sur ordinateur sont impressionnants, la discipline semble victime de son succès : on lui demande de prendre en compte des systèmes comprenant de plus en plus d'éléments, ces différents éléments sont de plus en plus en interaction les uns avec les autres, de multiples contraintes liées au fonctionnement viennent régir les variations des variables du système, ces systèmes peuvent être étudiés en régime permanent, mais également lors de régimes transitoires, et ce fonctionnement, permanent ou transitoire, fait souvent l'objet de l'optimisation d'un critère, puis l'objet d'un contrôle. Jusqu'à présent, ces défis ont été relevés en grande partie grâce aux méthodes du calcul numérique mises en œuvre sur ordinateurs. Proposant une approche différente, le calcul formel, ou symbolique, se propose de manipuler les modèles de façon formelle, retardant le plus possible l'évaluation des expressions mathématiques manipulées. De plus en plus souvent désormais, calcul formel et calcul numérique se voient associés dans le processus de modélisation et de simulation des systèmes. Le calcul formel constitue alors un préalable au calcul numérique, lui fournissant un modèle plus adapté aux traitements numériques que le modèle initial : il s'agit soit de transformer le modèle pour que sa structure corresponde à celle attendue par la méthode de résolution, soit de fournir des dérivées analytiques appréciées par cette méthode.

Le travail présenté suggère de pousser plus loin la collaboration entre calcul formel et calcul numérique en intégrant les manipulations symboliques, le calcul des dérivées notamment, aux algorithmes de résolution, jusqu'alors numériques. Plus précisément il s'agit, dans le contexte d'un environnement de modélisation et de simulation, donc dans un environnement de calcul numérique, d'appliquer systématiquement le principe du calcul formel : toute expression mathématique comprenant des symboles[1] doit être manipulée formellement. Dans ce cadre, l'évaluation numérique des expressions mathématiques a lieu le plus tard possible, c'est-à-dire uniquement lorsque l'algorithme de résolution la requiert. Cette approche symbolico-numérique

[1] Le terme « symbole » désigne ici un identifiant auquel n'est attachée aucune sémantique particulière a priori, par opposition aux variables du calcul numérique qui elles sont typées et valuées.

est plus particulièrement appliquée à la modélisation, la simulation et l'optimisation de systèmes continus. Sa mise en œuvre a donné lieu à une réalisation informatique : eXMSL, un environnement de résolution de problèmes, à l'aide duquel sont traités deux modèles. Les avantages attendus -capacité d'expression de modèles complexes, qualité numérique des grandeurs calculées, diminution du temps de simulation ou d'optimisation- sont confrontés aux résultats obtenus lors de ces études.

Le Chapitre 1 présente une analyse bibliographique, dont le but est d'établir les apports actuels du calcul formel dans les activités de simulation numérique en général. Les environnements de simulation métier font presque exclusivement appel aux techniques de calcul numérique. Les techniques du calcul formel lorsqu'elles interviennent, sont employées :

- soit conjointement avec des techniques de calcul numérique dans les systèmes de calcul formel ;
- soit lors d'une étape de prétraitement par un système de calcul formel, préalable à une simulation numérique.

La bibliographie recense les très rares cas où le calcul formel a investi la simulation numérique pour en améliorer la fiabilité, l'efficacité ou la facilité d'usage. Ce constat motive ce travail, qui pose la question de la coopération étroite entre un système de calcul formel et un système de calcul numérique.

Le Chapitre 2 spécifie un système de calcul symbolico-numérique pour la simulation de modèles continus. La spécification de ce système de calcul coopératif commence par la définition en 2.1 du sous-système en charge de construire et d'évaluer, formellement et/ou numériquement, les expressions mathématiques. Selon une analyse basée sur le langage UML (Object Management Group 2007), la partie 2.2 traite de la construction et de l'évaluation d'expressions mathématiques particulières : les modèles. Dans le cadre d'une simulation quantitative de systèmes continus, il s'agit de systèmes d'équations continues. La fiabilité apportée par une résolution symbolico-numérique de ces modèles est envisagée pour les systèmes d'équations linéaires, non linéaires, algébro-différentielles, ainsi que pour les problèmes d'optimisation sous contraintes non linéaires. Cette étude conduit à un modèle structurel et comportemental du sous-système dédié à la résolution symbolico-numérique d'équations continues. La partie 2.3 fait le choix d'une approche fonctionnelle des modèles : ceux-ci ne sont pas uniquement des systèmes d'équations continues, mais aussi des fonctions implicites définies par ces équations. L'application d'un théorème des fonctions implicites pour

la dérivation de ces différentes classes de modèles permet d'envisager un calcul fiable de leurs dérivées lors d'une optimisation ou d'une analyse de sensibilité.

Le Chapitre 3 présente une réalisation logicielle originale issue du modèle de conception élaboré Chapitre 1. Le système de calcul symbolico-numérique eXMSL reprend le modèle structurel et comportemental précédent, complété dans la partie 2.4 par des apports spécifiques visant une plus grande performance. Le modèle de conception ainsi enrichi est transformé en modèle d'implémentation pour donner lieu à une réalisation informatique effective. La démarche consistant à distinguer les modèles de conception et d'implémentation, et à définir une transformation pour produire le modèle d'implémentation à partir du modèle de conception et d'un modèle de la plate-forme informatique cible, s'inspire largement des principes de l'architecture dirigée par les modèles (Model Driven Architecture). La partie 3.1 détaille la transformation utilisée. La partie 3.2 donne les modalités informatiques d'accès au système de calcul symbolico-numérique produit : soit en tant que bibliothèque mathématique FORTRAN 90 pour un accès local, soit en tant que service Web pour un accès universel.

Deux applications du système de calcul symbolico-numérique eXMSL à des problèmes de génie des procédés, font l'objet du Chapitre 4. La partie 4.1 détaille la résolution d'un modèle statique, non linéaire et continu, le modèle de solvatation d'Engels. La partie 4.3 rend compte de la simulation d'un système dynamique hybride, la distillation de Rayleigh, abordé comme un système dynamique continu.

La conclusion synthétise les connaissances et les résultats acquis lors de ce travail. Elle dégage les points forts et les points faibles du modèle de coopération entre le calcul formel et le calcul numérique proposé, au travers du traitement de deux modèles particuliers. Plusieurs perspectives pour prolonger ce travail sont recensées. Deux d'entre elles, l'enrichissement du modèle de coopération et la simulation acausale de procédés, sont précisées.

Chapitre 1. Des collaborations entre calcul formel et calcul numérique

La simulation numérique sur ordinateur est aussi ancienne que l'informatique, les premiers ordinateurs ayant été conçus pour calculer des trajectoires de missiles. Avant même l'informatique de gestion ou l'informatique industrielle, l'informatique a d'abord été scientifique. Il est donc normal qu'une discipline vieille de plus de cinquante ans ait donné lieu à des résultats remarquables. Cela est tout particulièrement vrai pour la simulation numérique, au carrefour des mathématiques appliquées, et de l'informatique. Cette discipline a stimulé l'apparition et l'essor de méthodes numériques, comme les éléments finis, dont la mise en œuvre était facilitée par les progrès dans le domaine de l'architecture des ordinateurs ou des langages informatiques. Aussi aujourd'hui, existe-t-il une pléthore de logiciels de calcul scientifique, à vocation générale ou dédiés à un domaine particulier. La plupart de ces outils mettent en œuvre des algorithmes numériques. Certains incorporent, outre des techniques numériques, des techniques de calcul formel.

Cette première partie a pour but de clarifier la façon dont calcul formel et calcul numérique collaborent, ou pourraient collaborer, à la résolution de problèmes. Lorsqu'il s'agit d'employer les techniques du calcul formel et du calcul numérique pour simuler un système, il est possible de :

1. faire du calcul numérique à l'intérieur d'un système de calcul formel ;
2. utiliser le calcul formel comme préprocesseur pour le calcul numérique.

Chacun de ces choix comporte des avantages et des inconvénients, qui sont présentés. Au vu de cette analyse critique, nous proposons une approche coopérative innovante où des capacités de calcul formel sont adjointes aux environnements de calcul numériques usuels. Cette approche appliquée à la simulation et à l'optimisation de systèmes dynamiques continus sera l'objet du présent document.

1.1. Calcul numérique à l'intérieur d'un système de calcul formel

L'un des domaines d'application privilégiés des systèmes de calcul formel est… le calcul numérique ! En effet, du point de vue de l'ingénieur, ou du chercheur, le résultat attendu de la simulation d'un système physique, quand bien même celle-ci comprend une séquence de calculs symboliques, est une évaluation numérique. Les systèmes de calcul formel abordent donc la plupart des thématiques du calcul numérique telles que la représentation des nombres sur ordinateurs, les opérations élémentaires de l'algèbre linéaire, l'intégration numérique ou la résolution approchée de systèmes d'équations non linéaires ou différentielles. Le calcul flottant en précision arbitraire, assez rarement intégré dans les logiciels de calcul scientifique, est systématiquement présent dans les grands systèmes de calcul formel. Les nombres algébriques, racines de polynômes à coefficients entiers, apparaissent dans la plupart des calculs symboliques. Ils constituent une des spécificités des logiciels de calcul formel.

Les chapitres suivants donnent un aperçu de la façon dont le calcul numérique apparaît au sein des systèmes de calcul formel. Cette présentation n'est absolument pas exhaustive. Elle veut montrer à quel point la manipulation des symboles et la manipulation des nombres sont indissociables dès lors que l'on vise l'exactitude dans la résolution de problèmes mathématiques, ce qui est le cas du calcul formel.

1.1.1. Evaluation numérique d'expressions mathématiques

Tout système de calcul formel permet de manipuler des nombres et de construire des expressions mathématiques à partir de ces derniers, des opérateurs arithmétiques et des fonctions mathématiques usuelles. La syntaxe de ces expressions suit la plupart du temps les règles de construction habituelle : opérateurs unaires préfixés, opérateurs n-aires infixés, ordre de priorité des opérateurs prédéfini, … Dans ce domaine, les principales différences d'un logiciel à un autre sont relatives aux identifiants des fonctions mathématiques. Ainsi, sous Mathematica (Wolfram 2003) le reste de la division entière de n par m s'écrit-il **`Mod[n,m]`** alors qu'il s'exprime par **`rem(n,m)`** dans Axiom (Jenks & Sutor 1992). Une traduction des différents termes utilisés par les principaux systèmes de calcul formel est disponible sous le titre *« Rosetta Translations »* sur le site http://wiki.axiom-developer.org/RosettaStone.

Si des différences syntaxiques apparaissent bien souvent dès l'écriture d'expressions élémentaires, il est plus surprenant de constater également des différences sémantiques dans le traitement de ces mêmes expressions par différents systèmes. L'expression mathématique

$\cos(2{,}7) - \sin(3)\tan(\frac{7}{9})$ s'écrit **`Cos[2.7] - Sin[3] Tan[7/9]`** sous Mathematica, et **`cos(2.7) - sin(3) * tan(7/9)`** sous Maple. Bien que représentant la même expression, les deux commandes précédentes seront évaluées différemment au sein des deux systèmes. Mathematica retourne la valeur `-1.04306` tandis que Maple (Monagan et al. 2005a;Monagan et al. 2005b) retourne l'expression `-.9040721420-sin(3)*tan(7/9)`. Les deux outils évaluent la fonction cosinus au point 2,7 car il s'agit d'un nombre décimal, vu a priori comme une approximation d'un nombre réel. Maple maintient l'expression **`sin(3)*tan(7/9)`** non évaluée car **`3`** étant un entier et **`7/9`** un rationnel, l'évaluation sous la forme d'un nombre décimal conduirait à une perte de précision. Mathematica évalue la différence entre le nombre décimal, issu de l'évaluation de **`Cos[2.7]`**, et la quantité numérique **`Sin[3] Tan[7/9]`**, et produit un résultat approché sous la forme d'un nombre décimal. Cet exemple illustre en fait deux stratégies d'évaluation numérique diamétralement opposées[2] :

- le résultat de l'application d'une fonction, ou d'un opérateur numérique, à des arguments qui sont des quantités numériques, est un nombre décimal, approximation du résultat exact, si et seulement tous les arguments sont des nombres décimaux ;
- le résultat de l'application d'une fonction, ou d'un opérateur numérique, à des arguments qui sont des quantités numériques, est un nombre décimal, approximation du résultat exact, si et seulement l'un au moins des arguments est un nombre décimal.

En fait, la stratégie d'évaluation numérique adoptée caractérise particulièrement bien un système de calcul formel. Définir une telle stratégie, c'est entre autres :

1. choisir les types de données qui représenteront les nombres ;
2. choisir le moment de l'approximation numérique dans les algorithmes de calcul.

Ces deux choix étant au cœur du travail présenté, il convient de les illustrer dès à présent.

1.1.1.1. Types de données pour la représentation des nombres

Tout comme la plupart des langages informatiques, les systèmes de calcul formel organisent les nombres qu'ils sont susceptibles de manipuler par types. Ces types définissent à la fois la façon dont les nombres sont représentés sur ordinateur (nombre maximal de chiffres significatifs notamment), et les opérations applicables aux données de chaque type

[2] L'évaluation d'expressions numériques conduit plus généralement à des nombres complexes de la forme $a + b \cdot i$ où a et b sont des nombres décimaux.

(décomposition en nombres premiers pour un entier, partie imaginaire pour un nombre complexe, …). Les outils de calcul formel offrent cependant une palette de types pour classer les nombres, plus importante que les types primitifs de n'importe quel langage de programmation. Ils proposent bien évidemment un ou plusieurs types pour décrire les nombres entiers, les nombres réels et les nombres complexes ; tous proposent un ou plusieurs types pour décrire les nombres rationnels. Ainsi, le calcul dans le corps commutatif des nombres rationnels $(Q,+,\times)$ est-il exact. A titre d'exemple, l'évaluation de l'expression mathématique $3\cdot\left(\frac{7}{8}\right)^6+90$ à l'aide de Axiom, Maple, Mathematica ou Maxima, commandée par la saisie de **`3*(7/8)^6+90`** retourne la fraction entière $\frac{23945907}{262144}$. Une évaluation numérique de ce résultat est obtenue sous Mathematica à l'aide de la fonction **`N`** :

```
In[1]:= N[3*(7/8)^6+90]

Out[1]= 91.3464
```

Figure 1 – Approximation numérique d'une expression arithmétique sous Mathematica.

Un logiciel de calcul numérique, tel que MATLAB, retournera directement l'évaluation décimale `91.3464`. Cette distinction est primordiale, les nombres rationnels intervenant dans de multiples domaines, tels que les développements en séries entières ou les polynômes à coefficients entiers.

Sur le modèle de l'exemple précédent, il est toujours possible, au sein d'un système de calcul formel, d'évaluer une expression mathématique représentant un nombre, sous la forme d'un nombre décimal ou sous la forme d'un nombre complexe dont la partie réelle et la partie imaginaire sont des nombres décimaux. Cette évaluation numérique, qui est une approximation, est la plupart du temps entachée d'erreur. Sous Mathematica, la commande **`Sin[Pi/4]`** retourne la valeur exacte :

```
In[1]:= Sin[Pi/4]
```

Out[1]= $\frac{1}{\sqrt{2}}$

Figure 2 – Evaluation exacte d'une expression trigonométrique sous Mathematica.

L'évaluation numérique **N[Sin[Pi/4]]** retourne une approximation décimale du résultat exact :

```
In[1]:= N[Sin[Pi/4]]

Out[1]= 0.707107
```

Figure 3 – Approximation numérique d'une expression trigonométrique sous Mathematica.

L'évaluation numérique d'un nombre complexe produit un nombre complexe dont partie réelle et imaginaire sont des nombres décimaux :

```
In[1]:= N[Log[-5]]

     Out[1]= 1.60944+3.14159 i
```

Figure 4 – Approximation numérique d'un nombre complexe sous Mathematica.

1.1.1.2. Manipulations formelles versus approximations numériques

Le but du calcul formel est de transformer une expression mathématique initiale en une nouvelle expression rigoureusement équivalente, mais plus pertinente pour l'utilisateur de l'outil. Dès lors, les fonctions de résolution fournissent, lorsqu'elles le peuvent, un résultat exact et complet. La résolution de l'équation polynômiale d'une variable à coefficients entiers $x^3 - x + 7 = 0$ à l'aide de Mathematica permet d'obtenir l'expression exacte des trois racines :

```
In[1]:= Solve[x^3-x+7 == 0, x]
```

$$\text{Out[1]=}\ \left\{\left\{x \to -\left(\frac{2}{3(63-\sqrt{3957})}\right)^{1/3} - \frac{\left(\frac{1}{2}\left(63-\sqrt{3957}\right)\right)^{1/3}}{3^{2/3}}\right\},\ \left\{x \to \frac{\left(1+i\sqrt{3}\right)\left(\frac{1}{2}\left(63-\sqrt{3957}\right)\right)^{1/3}}{2\,3^{2/3}} + \frac{1-i\sqrt{3}}{2^{2/3}\left(3\left(63-\sqrt{3957}\right)\right)^{1/3}}\right\},\ \left\{x \to \frac{\left(1-i\sqrt{3}\right)\left(\frac{1}{2}\left(63-\sqrt{3957}\right)\right)^{1/3}}{2\,3^{2/3}} + \frac{1+i\sqrt{3}}{2^{2/3}\left(3\left(63-\sqrt{3957}\right)\right)^{1/3}}\right\}\right\}$$

Figure 5 – Evaluation exacte des racines d'un polynôme de degré trois sous Mathematica.

Une évaluation numérique de ces racines, peut être obtenue à partir du résultat de la commande précédente, désigné par **%** sous Mathematica :

```
In[2]:= N[%]

Out[2]=  {{x→-2.08675},{x→1.04337 -1.50528 i},{x→1.04337
+1.50528 i}}
```

Figure 6 – Approximation numérique des racines d'un polynôme de degré trois sous Mathematica.

Dans le cas présenté, le système de calcul formel procède à des manipulations formelles de l'expression mathématique initiale `Solve[x^3-x+7 == 0, x]` pour aboutir à l'expression mathématique finale : la liste des trois racines. Bien évidemment, au cours de ces manipulations formelles des calculs sur les nombres peuvent avoir lieu, mais Mathematica s'interdit toute approximation, telle qu'une approximation décimale de la racine carrée $\sqrt{3957}$. Le système procède à des manipulations formelles de bout en bout, à la fois sur les symboles et sur les nombres présents dans les expressions intermédiaires.

Formulons sous Mathematica le problème de la résolution de l'équation polynômiale $x^3 - x + 7 = 0$ de façon très légèrement différente, mais tout à fait équivalente du point de vue mathématique :

```
In[1]:= Solve[x^3-x+7.0 == 0, x]

Out[1]= {{x→-2.08675},{x→1.04337 -1.50528 i},{x→1.04337 +1.50528
i}}
```

Figure 7 – Approximation numérique des racines d'un polynôme de degré trois sous Mathematica.

Le résultat produit est désormais une approximation numérique de la racine réelle et des deux racines complexes, comme l'était le résultat du calcul de la Figure 6.

La formule de Cardan, appliquée à des coefficients dont l'un est un nombre décimal, a conduit à une succession d'approximations numériques, conduisant à l'estimation numérique de la valeur des racines. L'approximation initiale du nombre $\sqrt{3957.0}$ a conduit à une suite de calculs approchés, puis à un résultat approché. La valeur approchée des racines, affichée avec six chiffres significatifs, comporte en fait un nombre de chiffres significatifs plus important dépendant de la précision machine. La perte de précision associée à la spécification même du problème demeure irrécupérable.

Bien qu'exacte, l'expression des racines de la cubique trouvée à l'aide de la commande `Solve[x^3-x+7 == 0, x]` n'est pas très exploitable. Une évaluation numérique arbitrairement précise de cette expression est donnée par Mathematica, en précisant le nombre de chiffres significatifs souhaité, cinquante par exemple. Si `%` désigne l'expression `Solve[x^3-x+7 == 0, x]` :

```
In[2]:= N[%, 50]

Out[2]=
{{x→2.0867453398826663835823622043953787427080622113040},{x→1.04
337266994133319179118110219768937135403110565 20-
1.5052838885544210308373615429419731441434208915378
i},{x→1.0433726699413331917911811021976893713540311056520+1.5052
838885544210308373615429419731441434208915378 i}}
```

Figure 8 – Approximation numérique précise des racines d'un polynôme de degré trois sous Mathematica.

A l'inverse de la fonction **`Solve`**, qui privilégie l'obtention de solutions exactes, la fonction `NSolve` de l'environnement Mathematica recherche une approximation numérique des solutions :

```
In[1]:= NSolve[x^3-x+7 == 0, x]

Out[1]= {{x→-2.08675},{x→1.04337 -1.50528 i},{x→1.04337 +1.50528
i}}
```

Figure 9 – Calcul numérique des racines d'un polynôme de degré trois sous Mathematica.

Dans ce cas, **NSolve** applique la formule de Cardan à partir des coefficients de l'équation polynômiale, considérés du point de vue informatique comme des nombres décimaux et non comme des nombres entiers. Dès lors, les opérations arithmétiques présentes dans cette formule sont toutes exécutées en calcul flottant, avec les erreurs liées à ce mode d'évaluation. L'option **WorkingPrecision** de la fonction **NSolve** permet toutefois d'accéder à un

```
In[1]:= NSolve[x^3-x+7 == 0, x, WorkingPrecision -> 50]

Out[1]=
{{x→2.0867453398826663835823622043953787427080622113040},{x→1.04
337266994133319179118110219768937135403110565 20-
1.5052838885544210308373615429419731441434208915378
i},{x→1.0433726699413331917911811021976893713540311056520+1.5052
838885544210308373615429419731441434208915378 i}}
```

nombre de chiffres significatifs plus important, cinquante par exemple :

Figure 10 – Calcul numérique précis des racines d'un polynôme de degré trois sous Mathematica.

Quelle que soit la précision demandée, le processus de résolution adopté par la fonction `NSolve`, lors de la résolution de l'équation $x^3 - x + 7 = 0$, peut être vu comme la succession d'une étape de manipulation formelle et de plusieurs étapes d'approximations numériques.

L'analyse d'un exemple particulier, traité à l'aide d'un système de calcul formel donné, nous conduit à trois remarques de portée pourtant beaucoup plus générale :

1. les systèmes de calcul formels procèdent par étapes, étapes de manipulations formelles ou étapes d'approximations numériques ;
2. une succession d'étapes de manipulations formelles conduit à un résultat exact. L'approximation numérique ultérieure du résultat exact sera au moins aussi précise qu'une approximation numérique obtenue par combinaison d'étapes de manipulations formelles et d'étapes d'approximations numériques ;
3. les types informatiques (entiers, décimaux, rationnels ou autres) des expressions numériques apparaissant dans les expressions mathématiques transformées dictent, en partie, la part

relative et l'ordre des étapes de manipulations formelles et des étapes d'approximations numériques.

De plus comme cela a déjà été illustré par l'exemple de l'évaluation de $\cos(2{,}7)-\sin(3)\tan(\frac{7}{9})$:

4. chaque système de calcul formel choisit des stratégies de transformation d'expressions, caractérisées notamment par des successions d'étapes de manipulations formelles et d'approximations numériques.

Ces remarques préfigurent déjà le thème qui sera discuté dans ce document : la conception d'un système coopératif intégrant calcul formel et calcul numérique. Cette conception passera nécessairement par des choix relatifs à :

- la précision des résultats recherchés ;
- les types de données pour la représentation des nombres ;
- la stratégie d'évaluation.

1.1.2. Calcul à précision multiple

La question de l'exactitude des calculs numériques dans les logiciels scientifiques est un vaste sujet de recherche né du fait que l'arithmétique des ordinateurs est une arithmétique bien spécifique, travaillant sur des mots de longueur donnée, où l'information est codée dans une base spécifique, la base deux. En aucun cas, arithmétique usuelle et arithmétique des ordinateurs ne se confondent !

Dans le cas particulier des systèmes de calcul formel, on distinguera l'arithmétique rationnelle et l'arithmétique réelle. Les calculs sur les nombres rationnels y sont exacts. Les calculs sur les nombres réels, bien qu'approchés, peuvent être effectués en précision multiple.

1.1.2.1. Arithmétique rationnelle exacte

Essentielle dans un système de calcul formel, l'arithmétique rationnelle y est exacte. Ainsi, sous Mathematica, la somme partielle $\sum_{i=1}^{n}\frac{1}{i}$ peut-elle être calculée de manière exacte pour n aussi grand que souhaité, sous réserve de disposer de suffisamment de mémoire vive pour ce calcul :

```
In[1]:= Sum[1/i, {i,1,100}]
```

Out[1]= $\frac{14466636279520351160221518043104131447711}{2788815009188499086581352357412492142272}$

Figure 11 – Un calcul en arithmétique rationnelle exacte sous Mathematica.

En pratique, les systèmes de calcul formel définissent un type de données particulier pour manipuler les rationnels, et considèrent ceux-ci comme le quotient de deux nombres entiers de longueurs arbitraires. La fraction est évidemment simplifiée si possible.

1.1.2.2. Arithmétique réelle approchée

L'un des problèmes majeurs du calcul numérique sur ordinateurs est l'erreur d'arrondi souvent commise lors de la représentation d'un nombre réel sur ordinateur. Le nombre maximum de chiffres significatifs représentables pour un nombre réel étant fixé a priori pour une architecture informatique donnée, cette erreur a lieu dès lors que le nombre de chiffres significatifs de la valeur à représenter excède cette limite. Pire encore, la densité de représentation des nombres réels n'étant pas uniforme -celle-ci est plus dense au voisinage de zéro et diminue lorsque la valeur absolue du nombre représenté augmente-, cette erreur d'arrondi est difficile à maîtriser au cours des calculs. Cette difficulté, propre au calcul numérique sur ordinateur se retrouve bien évidemment dans les systèmes de calcul formel, lorsque ceux-ci travaillent sur les nombres réels en précision machine. Une illustration simple de la propagation des erreurs d'arrondi, et de ses conséquences fâcheuses, est le calcul d'expressions mal conditionnées, i.e. d'expressions pour lesquelles les sous-expressions ont des ordres de grandeur très différents. Tel est le cas par exemple lors de la séquence de calcul suivante : $x = 10.0^{50}$, $y = x + 1$, $z = \frac{1}{y - x}$ où x, y, et z sont des nombres réels représentés en virgule flottante en double précision au standard IEEE/754. L'interprétation de ces instructions par le système Mathematica prend la forme suivante :

```
In[1]:= x=10.^50

Out[1]= 1.×10^50

In[2]:= y=x+1

Out[2]= 1.×10^50

In[3]:= z=1/(y-x)

Power ::infy : Infinite expression 1/0. encountered . Plus...

Out[3]= ComplexInfinity
```

Figure 12 – Propagation des erreurs d'arrondi lors d'un calcul numérique mal conditionné sous Mathematica.

La valeur de y après son affectation est 10^{50} car, en double précision, il est impossible de stocker les cinquante et un chiffres significatifs du résultat exact. La division $\frac{1}{y-x}$ conduit donc à l'exception *« division par zéro »*.

La solution la plus simple apportée aux problèmes induits par les erreurs d'arrondi est l'utilisation d'une arithmétique ayant une plus grande précision, c'est-à-dire représentant les nombres réels avec un plus grand nombre de chiffres significatifs, et dont les fonctions élémentaires manipulent un plus grand nombre de chiffres significatifs. Ainsi, dans un langage de programmation proposant plusieurs sous-types pour déclarer des nombres réels, un algorithme écrit en simple précision pourra-t-il être repris en double précision, voire en quadruple précision si celle-ci est disponible. Malgré tout, la plupart du temps, il est impossible de connaître la précision requise pour mener à bien une séquence de calculs. Et la plupart des *« propriétés intuitives »* de l'arithmétique en virgule flottante sont battues en brèche par de nombreux exemples. Au rang des *« propriétés intuitives »*, (Ménissier-Morain 2005) cite :

1. même si le résultat n'est pas rigoureusement exact, il est certainement proche du résultat exact, c'est-à-dire que les erreurs d'arrondi sont mineures et qu'en particulier l'ordre de grandeur est supposé être préservé ;
2. un nombre limité d'opérations en virgule flottante engendre uniquement une légère inexactitude du résultat calculé ;
3. le mode d'arrondi a peu d'effet sur la qualité du résultat ;
4. si un résultat est calculé avec plus de chiffres, son exactitude sera meilleure. Plus précisément, nous espérons que la distance entre la valeur calculée et la valeur réelle décroît avec l'augmentation du nombre de chiffres utilisés dans les calculs ;
5. le résultat ne dépend pas de l'arithmétique, de l'ordinateur, etc.

Un contre-exemple des propriétés intuitives 1. et 2. est le calcul de la différence $e^{\pi\sqrt{163}} - 262537412640768744$. Comme le soulignent (Gomez & Quadrat 1991), le résultat exact de ce calcul a pour valeur approchée : $-7{,}5\cdot 10^{-13}$. Une évaluation en double précision produit un résultat d'un tout autre ordre de grandeur : -480. La ligne suivante retranscrit ce calcul effectué sous Mathematica en précision machine :

```
In[1]:= N[Exp[Pi Sqrt[163]] - 262537412640768744]

Out[1]= -480.
```

Figure 13 – Un calcul numérique en précision machine produisant un résultat complètement erroné sous Mathematica.

La demande d'une précision de dix chiffres sur le résultat du calcul permet d'obtenir un résultat beaucoup plus exact :

```
In[1]:= N[Exp[Pi Sqrt[163]] - 262537412640768744, 10]

Out[1]= -7.499274028×10^-13
```

Figure 14 – Un calcul numérique pour lequel une précision de dix chiffres est demandée à Mathematica.

Face aux écueils du calcul en précision donnée, la plupart des systèmes de calcul formel - Axiom, Mathematica, Maxima, MuPAD (Creutzig & Oevel 2004), PARI/GP et Reduce- ont adopté l'arithmétique à virgule flottante en précision arbitraire, désignée également par le terme de *« calcul à précision multiple »*. Cette arithmétique tente dans le calcul précédent d'obtenir un résultat comportant dix chiffres significatifs exacts. Dans le cadre de cette arithmétique, un nombre réel est représenté par une séquence de chiffres significatifs arbitrairement longue et l'information relative à sa précision. Lors de tout calcul, le système de calcul formel essaye de fournir un résultat ayant la plus grande précision possible, compte tenu de la précision des nombres présents dans l'expression mathématique. Pour cela, la précision du résultat de tout calcul intermédiaire est évaluée en utilisant un modèle de propagation de l'erreur spécifique.

1.1.2.3. Une arithmétique particulière de calcul à précision multiple : Mathematica

Même si tous les systèmes de calcul formel renommés proposent le calcul à précision multiple, la mise en œuvre pratique de cette arithmétique diffère d'un outil à un autre. Récemment, les détails de la réalisation de l'arithmétique à virgule flottante en précision arbitraire dans Mathematica ont été publiés par (Sofroniou & Spaletta 2005). Ce chapitre résume cet article, notamment du point de vue du modèle de propagation de l'erreur adopté, afin de montrer les intérêts et les limites du calcul à précision multiple.

L'arithmétique réelle dans Mathematica s'appuie sur une interprétation préalable des nombres décimaux saisis par l'utilisateur. Suite à la lecture du nombre décimal x, le système détermine le nombre de chiffres significatifs dans x, et estime l'ordre de grandeur de l'erreur absolue associée à x : $-\log_{10}(|\Delta_x|)$, ainsi que l'ordre de grandeur de l'erreur relative associée à x : $-\log_{10}(\left|\frac{\Delta_x}{x}\right|)$. En pratique, Mathematica propose trois fonctions **`Scale`**, **`Accuracy`** et

Precision, pour accéder respectivement à l'ordre de grandeur d'un nombre x, à l'ordre de grandeur de l'erreur absolue associée à x et à l'ordre de grandeur de l'erreur relative associée à x. Etant donné un nombre non nul x, les valeurs de ces trois fonctions en ce point sont reliées par l'égalité suivante, exprimée dans la syntaxe de Mathematica :

```
Precision[x] == Scale[x] + Accuracy[x]
```

Équation 1 – Relation entre précision, ordre de grandeur et exactitude gouvernant le modèle de calcul à précision multiple de Mathematica.

L'ordre de grandeur, la précision et l'exactitude de chaque nombre décimal étant connus au sens où ces quantités ont été définies précédemment, Mathematica définit un modèle de propagation de l'erreur dans les calculs faisant intervenir des nombres décimaux. Dans une certaine mesure, ce modèle s'apparente au modèle de propagation de l'erreur dans l'arithmétique à intervalles, présentée par exemple dans (Nickel 1985). Ainsi, à tout nombre décimal x, pour lequel l'ordre de grandeur de l'erreur absolue est $-\log_{10}(|\Delta_x|)$, est associé un intervalle $\left[x-\frac{|\Delta_x|}{2}, x+\frac{|\Delta_x|}{2}\right]$ contenant nécessairement la valeur exacte de x. Si l'on considère un autre nombre décimal y appartenant à l'intervalle $\left[y-\frac{|\Delta_y|}{2}, y+\frac{|\Delta_y|}{2}\right]$, la somme des deux intervalles $\left[x+y-\frac{|\Delta_x|}{2}-\frac{|\Delta_y|}{2}, x+y+\frac{|\Delta_x|}{2}+\frac{|\Delta_y|}{2}\right]$, telle qu'elle est définie dans l'arithmétique d'intervalle, contient la valeur exacte de $x+y$. De plus, l'ordre de grandeur de l'erreur absolue associée à $x+y$ est $-\log_{10}(|\Delta_x|+|\Delta_y|)$. Il en résulte que **Accuracy[x+y]** s'exprime directement à partir des quantités **Accuracy[x]** et **Accuracy[y]** :

$$Accuracy[x+y] = -\log_{10}\left(10^{-Accuracy[x]} + 10^{-Accuracy[y]}\right)$$

Precision[x+y] peut être déduit de Équation 1. Hélas, la propagation de l'erreur ne peut pas être quantifiée de façon aussi exacte dans la plupart des cas. Si l'on considère les nombres décimaux x et y, respectivement associés aux intervalles $\left[x(1-\frac{|\Delta_x|}{2x}), x(1+\frac{|\Delta_x|}{2x})\right]$ et $\left[y(1-\frac{|\Delta_y|}{2y}), y(1+\frac{|\Delta_y|}{2y})\right]$, le produit des deux intervalles, contenant la valeur exacte du produit $x \cdot y$, s'écrit :

$$[\min(xy(1-\frac{|\Delta_x|}{2x})(1-\frac{|\Delta_y|}{2y}), xy(1-\frac{|\Delta_x|}{2x})(1+\frac{|\Delta_y|}{2y}), xy(1+\frac{|\Delta_x|}{2x})(1-\frac{|\Delta_y|}{2y}), xy(1+\frac{|\Delta_x|}{2x})(1+\frac{|\Delta_y|}{2y})),$$
$$\max(xy(1-\frac{|\Delta_x|}{2x})(1-\frac{|\Delta_y|}{2y}), xy(1-\frac{|\Delta_x|}{2x})(1+\frac{|\Delta_y|}{2y}), xy(1+\frac{|\Delta_x|}{2x})(1-\frac{|\Delta_y|}{2y}), xy(1+\frac{|\Delta_x|}{2x})(1+\frac{|\Delta_y|}{2y}))]$$

Afin de pouvoir exprimer aisément les quantités **`Precision[x.y]`** et **`Accuracy[x.y]`** à partir des quantités élémentaires **`Precision[x]`**, **`Precision[y]`**, **`Accuracy[x]`** et **`Accuracy[y]`**, Mathematica néglige les termes du second ordre $\frac{|\Delta_x||\Delta_y|}{4}$, et considère que la valeur exacte du produit $x \cdot y$ appartient à l'intervalle $\left[xy(1-\frac{|\Delta_x|}{2x}-\frac{|\Delta_y|}{2y}), xy(1+\frac{|\Delta_x|}{2x}+\frac{|\Delta_y|}{2y})\right]$. La quantité **`Precision[x.y]`** est dès lors calculée à partir des valeurs de **`Precision[x]`** et **`Precision[y]`** :

$$\Pr ecision[x \cdot y] = -\log_{10}\left(10^{-\Pr ecision[x]} + 10^{-\Pr ecision[y]}\right)$$

`Accuracy[x.y]` peut être déduit de Équation 1.

Le modèle de propagation de l'erreur adopté par ce système particulier est donc approché. Malgré cette inexactitude relative, ce modèle n'a pas les inconvénients souvent liés à l'arithmétique à intervalles qui peut fournir des bornes sur l'erreur très pessimistes. Pour la plupart des fonctions autres que l'addition et la multiplication, le modèle de propagation de l'erreur est basé sur la linéarisation de la fonction concernée au point de calcul. A titre d'exemple, la linéarisation de la fonction exponentielle au point de calcul x permettra d'approcher l'erreur absolue $\Delta_{\exp(x)}$ associée à $\exp(x)$, en fonction de l'erreur absolue Δ_x associée à x :

$$\frac{\Delta_{\exp(x)}}{\exp(x)} \approx \Delta_x$$

Mathematica traduit cette approximation par la relation :

`Precision[Exp[x]]==Accuracy[x]`

`Accuracy[Exp[x]]` est déduit de la relation :

`Precision[Exp[x]]==Scale[Exp[x]]+Accuracy[Exp[x]]`

Le modèle d'erreur dans Mathematica est donc basé sur des principes simples, systématiques et éprouvés -l'arithmétique de signifiance a vu le jour à la fin des années cinquante-, qui le rendent performant. Intégré de manière transparente à la représentation des nombres réels en précision multiple, il rend aisée toute analyse d'erreur. Il faut toutefois conserver à l'esprit que ce modèle d'erreur est un modèle approché, donc faux, constituant un compromis entre performance et fiabilité.

D'autres approches pour le calcul à précision multiple existent dans les autres systèmes de calcul formel. La plupart du temps elles se limitent à proposer de choisir le nombre de chiffres significatifs utilisé pour représenter les nombres décimaux tout au long d'un calcul numérique. Tel est le cas par exemple dans Maple, où la variable d'environnement **`Digits`** détermine ce nombre de chiffres significatifs. A tout moment il est possible de modifier cette variable, mais en aucun cas la précision des calculs ne s'adapte aux caractéristiques des arguments ou de la fonction évaluée. Contrairement à Mathematica, les autres systèmes de calcul formel n'intègrent pas un modèle d'erreur complet permettant de :

- demander a priori une précision pour le résultat d'un calcul ;
- juger a posteriori de la qualité du résultat produit.

1.1.2.4. Vers une arithmétique réelle exacte ?

Même si les systèmes de calcul formel font de leur mieux lors du calcul à précision multiple, en approchant avec une plus grande précision les résultats intermédiaires, l'exactitude des résultats numériques produits n'y est pas garantie. Comme on l'a vu au chapitre précédent, la raison principale en est l'absence d'un modèle d'erreur, ou la mise en œuvre d'un modèle d'erreur inexact, même s'il se révèle satisfaisant dans la plupart des cas.

Le modèle de propagation d'erreur adopté aujourd'hui en calcul numérique est fourni par l'arithmétique d'intervalles. Il s'agit de propager lors des calculs en nombres décimaux une borne supérieure de l'erreur d'arrondi commise lors de chaque opération. A la fin du calcul, on dispose donc du résultat calculé et d'une borne supérieure de l'erreur d'arrondi associée à ce résultat. Lorsque l'ordre de grandeur de cette borne est comparable ou supérieur à l'ordre de grandeur du résultat, le résultat ne peut pas être considéré comme exact avec certitude. Lorsque l'ordre de grandeur de cette borne est très inférieur à l'ordre de grandeur du résultat, le résultat peut être considéré comme exact.

MuPAD est l'un des rares systèmes de calcul formel à avoir intégré en partie l'arithmétique à intervalles, en proposant notamment la conversion de nombreuses expressions

arithmétiques sous forme d'intervalles à l'aide de la fonction `interval`. Si l'arithmétique d'intervalles n'est pas disponible dans les systèmes de calcul formel par défaut, des modules additionnels ont été développés par ailleurs. Ainsi intpakX permet-il aux utilisateurs de Maple de disposer d'une arithmétique à intervalles en précision multiple.

L'arithmétique à intervalles indique quand le résultat d'un calcul est faux. En cela, il s'agit d'un progrès par rapport à l'arithmétique en virgule flottante, ou par rapport au calcul à précision multiple, qui n'informent en rien sur l'erreur d'arrondi associée au résultat. Toutefois, l'objectif du calcul numérique reste bien évidemment d'obtenir systématiquement un résultat exact, ou aussi proche du résultat exact que nous le souhaitons. Comme le souligne (Revol & Rouillier 2007), les limites d'application de certains algorithmes numériques sont désormais atteintes du fait du nombre considérable d'opérations effectuées sur les modèles actuels. Une réponse actuelle est l'arithmétique à intervalles à précision multiple, telle qu'elle est mise en œuvre dans le module intpakX pour Maple, ou dans la bibliothèque MPFI. Une autre réponse est d'éviter les difficultés inhérentes à toute arithmétique en virgule flottante, aussi précise soit elle, et de proposer une *« arithmétique réelle exacte »*. Le but est de maintenir une borne supérieure unique pour toutes les erreurs d'arrondi commises au cours d'un calcul, même si cela requiert une très grande précision à certaines étapes du calcul. Le moyen est l'utilisation exclusive de nombres entiers, soit au travers de suites de Cauchy, soit au travers de fractions continues, aussi proches que voulu du nombre réel à représenter. Une réalisation particulièrement intéressante de cette arithmétique réelle exacte est celle décrite dans (Ménissier-Morain 2005). Les nombres réels calculables sont représentés comme des suites de nombres B-adiques finis. Pour chaque fonction, rationnelle, algébrique ou transcendante, un algorithme de calcul décrit comment produire la suite représentant le résultat de l'application de cette fonction à ses arguments, en fonction des suites représentant ces arguments. Pour chaque algorithme, il est prouvé que la suite obtenue est une représentation valide du résultat réel exact. Cette arithmétique est la première arithmétique réelle dont les algorithmes soient prouvés mathématiquement corrects. Ce travail a donné lieu à une réalisation au sein du langage de programmation Caml.

D'autres réalisations d'arithmétique réelle exacte sont disponibles, telles que iRRAM (Müller 2001) ou MPFR (Fousse et al. 2007), mais aucun système de calcul formel n'intègre aujourd'hui ces techniques. Cela est certainement temporaire car les outils de calcul formel gagneraient à disposer d'un traitement numérique exact, et les deux conditions préalables à la mise en œuvre d'une arithmétique réelle exacte -les entiers arbitrairement grands et l'arithmétique rationnelle exacte- sont disponibles au sein de tout système de calcul formel.

1.1.3. Nombres algébriques

Outre les nombres entiers et les nombres rationnels, les systèmes de calcul formel manipulent des nombres complexes, parmi lesquels les nombres algébriques. Ces nombres, racines de polynômes à coefficients entiers[3], jouent un rôle fondamental en arithmétique où leurs propriétés sont étudiées. En calcul formel, ils apparaissent dans la plupart des calculs dès lors que l'on s'abstient de toute évaluation numérique approchée. Ainsi $\sqrt{2}$ est-il un nombre algébrique car c'est une racine du polynôme X^2-2. Le nombre complexe i, racine du polynôme X^2+1, est également un nombre algébrique. Dans le domaine du calcul scientifique sur ordinateurs, seuls les systèmes de calcul formel représentent et manipulent ces nombres de façon particulière, afin notamment de pouvoir tester leur égalité. Après avoir discuté de la représentation des nombres algébriques dans différents systèmes de calcul formel, nous nous intéressons à la manipulation de ceux-ci.

1.1.3.1. Représentation

Tout nombre entier n est évidemment un nombre algébrique, puisqu'il est racine du polynôme à coefficients entiers $X-n$. Tout nombre rationnel $\frac{a}{b}$ est un nombre algébrique, puisqu'il est racine du polynôme à coefficients entiers $aX-b$. Tout nombre réel $\sqrt{n}$, où $n \in \mathrm{N}$, est un nombre algébrique, puisqu'il est racine du polynôme à coefficients entiers X^2-n . Il existe toutefois des nombres algébriques qu'il est impossible d'exprimer avec des radicaux. Les racines du polynôme X^5-X+1 en constituent un exemple. La résolution de l'équation $X^5-X+1=0$ à l'aide de Mathematica fournit le résultat suivant :

[3] On montre que si un nombre complexe est racine d'un polynôme dont les coefficients sont des nombres algébriques, alors ce nombre est algébrique.

```
Nombres algébriques.nb
In[1]:= Solve[X^5 - X + 1 == 0, X]
Out[1]= {{X → Root[1 - #1 + #1^5 &, 1]}, {X → Root[1 - #1 + #1^5 &, 2]},
         {X → Root[1 - #1 + #1^5 &, 3]}, {X → Root[1 - #1 + #1^5 &, 4]}, {X → Root[1 - #1 + #1^5 &, 5]}}
In[2]:= N[%]
Out[2]= {{X → -1.1673}, {X → -0.181232 - 1.08395 i},
         {X → -0.181232 + 1.08395 i}, {X → 0.764884 - 0.352472 i}, {X → 0.764884 + 0.352472 i}}
100%
```

Figure 15 – Représentation des nombres algébriques constituant les racines d'un polynôme sous Mathematica.

Dans ce cas, les nombres algébriques sont représentés en tant que racines de l'équation $X^5 - X + 1 = 0$. Un numéro d'ordre précise de quelle racine il s'agit sachant que Mathematica adopte les conventions suivantes :

- les racines réelles sont représentées avant les racines complexes ;
- les paires de racines complexes conjuguées sont adjacentes.

Une évaluation numérique des nombres algébriques à l'aide de la fonction `N` nous informe que le polynôme considéré admet une racine réelle et deux paires de racines complexes conjuguées. De façon similaire, Maple décrit les racines de l'équation précédente de manière implicite, mais adopte des conventions différentes pour l'ordre de représentation de ces racines : celles-ci sont ordonnées en fonction de la valeur de leur module et de leur argument.

```
Maple 9.5 - D:\Mes documents\Recherche\Thèse\Calcul formel\Maple\Nombres algébriques.mw - [Server 1]
File Edit View Insert Format Tools Window Help
Expression
Symbol
Matrix
Vector
> solve(X^5-X+1=0,X);
RootOf(_Z^5 - _Z + 1, index = 1), RootOf(_Z^5 - _Z + 1, index = 2),
    RootOf(_Z^5 - _Z + 1, index = 3), RootOf(_Z^5 - _Z + 1, index = 4),
    RootOf(_Z^5 - _Z + 1, index = 5)
> evalf(%);
0.7648844336 + 0.3524715460 I, -0.1812324445 + 1.083954101 I, -1.167303978,
    -0.1812324445 - 1.083954101 I, 0.7648844336 - 0.3524715460 I
```

Figure 16 – Représentation des nombres algébriques constituant les racines d'un polynôme sous Maple.

Evidemment le polynôme donné en argument des fonctions **`Root`** ou **`RootOf`** de Mathematica ou Maple est un polynôme irréductible. Ainsi, les solutions de l'équation $10X^5 - 10X + 10 = 0$ sont-elles exprimées en fonction des racines du polynôme irréductible $X^5 - X + 1$:

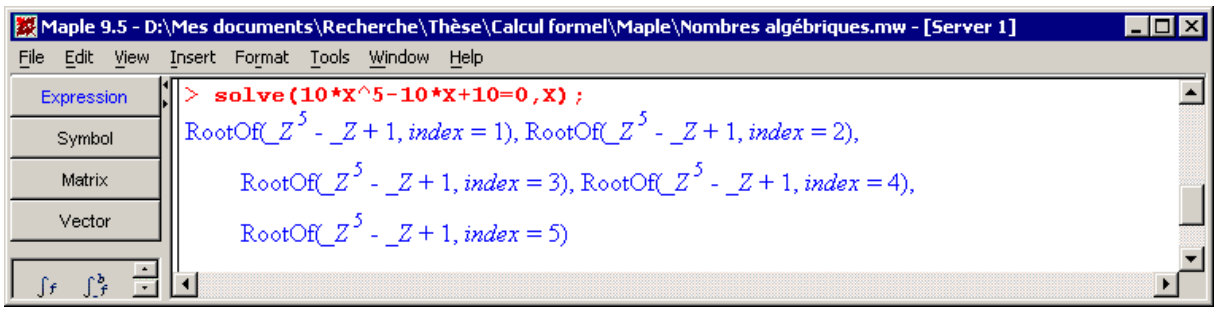

Figure 17 – Utilisation d'un polynôme irréductible lors de la représentation des nombres algébriques constituant les racines d'un polynôme sous Maple.

L'approche adoptée par Axiom dans la manipulation des nombres algébriques est différente de celle proposée par la plupart de ses concurrents. La résolution de l'équation $10X^5 - 10X + 10 = 0$ retourne l'équation équivalente $X^5 - X + 1 = 0$, faisant apparaître le polynôme irréductible. L'ensemble des solutions n'est pas énuméré :

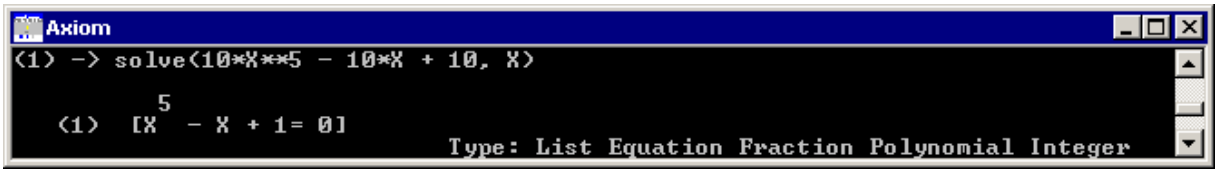

Figure 18 – Représentation des racines d'un polynôme par un polynôme irréductible sous Axiom.

Le résultat de l'application de la fonction **`rootOf`** au polynôme irréductible représente dans Axiom l'une quelconque des racines de ce polynôme, et non plus une racine particulière :

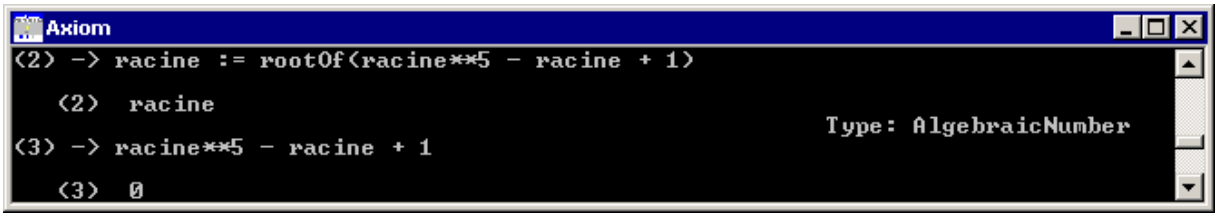

Figure 19 – Désignation générique d'une racine d'un polynôme sous Axiom, à l'aide de la fonction **`rootOf`**.

Dans le cas où certaines racines d'une équation polynômiale peuvent s'exprimer en fonction de radicaux, la fonction **`radicalSolve`** d'Axiom donne leur expression explicite exacte :

```
Axiom
(1) -> radicalSolve(10*X**5 - 10*X + 10, X)

   (1)  []
                                          Type: List Equation Expression Integer
(2) -> radicalSolve(X**3 - 3*X + 1 = 0, X)

   (2)
                      +----------+2                          +----------+2
                      | +---+                                | +---+
          +---+       |\|- 3  - 1                  +---+     |\|- 3  - 1
     (- \|- 3  + 1) 3|----------  - 2       (- \|- 3  - 1) 3|----------  + 2
                      \|     2                              \|     2
   [X= ------------------------------, X= ------------------------------,
                        +----------+                           +----------+
                        | +---+                                | +---+
            +---+       |\|- 3  - 1                  +---+     |\|- 3  - 1
        (\|- 3  + 1) 3|----------              (\|- 3  - 1) 3|----------
                        \|     2                              \|     2
         +----------+2
         | +---+
         |\|- 3  - 1
       3|----------   + 1
       \|     2
    X= ------------------]
            +----------+
            | +---+
            |\|- 3  - 1
          3|----------
          \|     2
                                          Type: List Equation Expression Integer
```

Figure 20 – Expression des racines d'un polynôme à l'aide de radicaux sous Axiom, à l'aide de la fonction **`radicalSolve`**.

1.1.3.2. Manipulation

Comme le souligne (Recio & Gonzalez-Lopez 1998), la plupart des systèmes de calcul formel disponibles semblent incapables de définir correctement les nombres algébriques et de les manipuler au moyen des opérateurs arithmétiques ou des opérateurs de comparaison. Selon eux, la principale raison en est le fait que les racines conjuguées d'un polynôme ne sont pas isolées par ces outils. L'insuffisance dans le traitement des nombres algébriques est illustrée par l'incapacité de Maple par exemple à décider de la valeur du prédicat $\sqrt{2} > -\sqrt{2}$:

```
Maple 9.5 - D:\Mes documents\Recherche\Thèse\Calcul formel\Maple\Nombres algébriques.mw - [Server 1]
File  Edit  View  Insert  Format  Tools  Window  Help
Expression
Symbol
Matrix
Vector
> evalb(5 > 3);
                                        true
> evalb(sqrt(2) > -sqrt(2));
                                        -√2 < √2
```

Figure 21 – Echec de la comparaison de deux nombres algébriques sous Maple.

(Rioboo 2003) rappelle l'égalité $\sqrt[3]{-\sqrt[5]{\frac{27}{5}}+\sqrt[5]{\frac{32}{5}}} = (-\sqrt[5]{3^2}+\sqrt[5]{3}+1)\sqrt[5]{\frac{1}{25}}$ proposée par Ramanujan, citée dans (Davenport, Siret, & Tournier 1987), qu'il est impossible de vérifier formellement sous Mathematica. Seule une évaluation numérique à une certaine précision de leur différence permet de supputer que les deux nombres sont égaux :

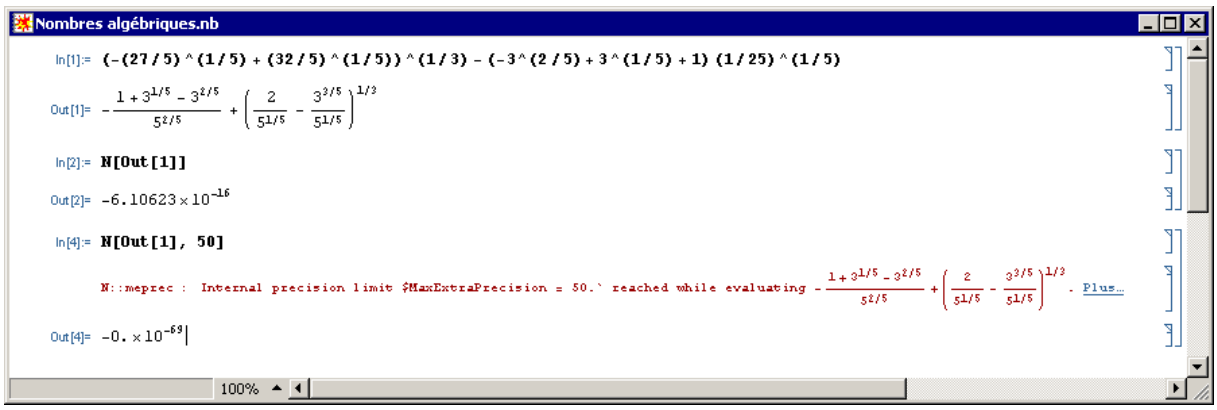

Figure 22 – Echec de la simplification d'une différence de deux nombres algébriques égaux sous Mathematica.

Dans (Gräbe 1999), Hans-Gert Gräbe dresse un bilan de la simplification par différents systèmes de calcul formel d'expressions contenant des radicaux imbriqués. Une des expressions choisies est $\sqrt{-6\sqrt{2}+11}+\sqrt{6\sqrt{2}+11}$ qui vaut 6. Si Derive et Maple parviennent à un résultat correct automatiquement, Maxima, Mathematica et MuPAD requièrent l'application d'une fonction de simplification spécifique, tandis que Axiom et Reduce laissent l'expression initiale inchangée. Ceci est illustré au travers des instructions Mathematica suivantes :

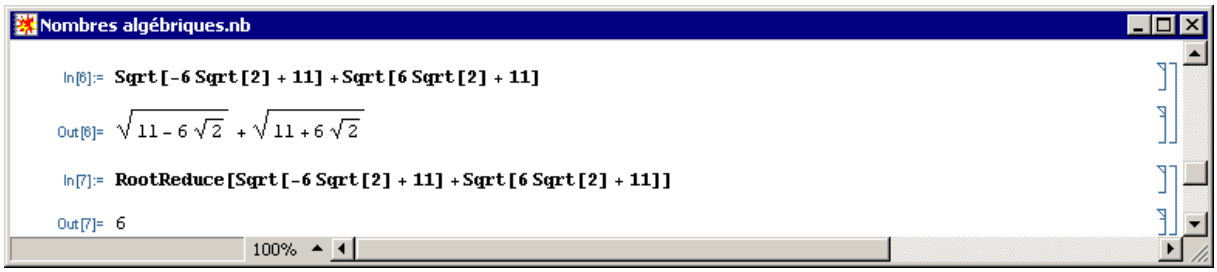

Figure 23 – Recours à la fonction **`RootReduce`** de Mathematica pour la simplification d'une somme de deux nombres algébriques.

Une autre expression choisie est $\sqrt[3]{6\sqrt{3}-10}+\sqrt[3]{6\sqrt{3}+10}$ qui vaut $2\sqrt{3}$. Seul Derive évalue correctement cette expression. L'auteur en conclut que pour les nombres algébriques admettant une représentation à l'aide de radicaux, ces représentations ne sont pas adaptées au calcul formel. En effet, lorsqu'elles sont utilisées ces représentations ne permettent pas de décider automatiquement et systématiquement si les nombres algébriques représentés sont égaux à zéro. On leur préfèrera donc un formalisme proposé par la plupart des systèmes de calcul formel et présenté plus haut : une racine particulière, ou générique, d'un polynôme P sera représentée par l'application d'une fonction -**`Root`** ou **`RootOf`**- au polynôme irréductible Q associé à P. Ce formalisme semble plus adapté à la manipulation de nombres algébriques au sein d'expressions mathématiques. Cependant (Recio & Gonzalez-Lopez 1998) note que les résultats d'opérations algébriques sur ces représentations ne sont pas très probants. Ils soulignent que si un nombre algébrique est déterminé par un polynôme, dont il est une racine, et une

information supplémentaire permettant de préciser la racine parmi les racines conjuguées, alors toute la difficulté est de développer des algorithmes pour les opérations arithmétiques sur ces nombres codés. Dans les systèmes de calcul formel actuels, ces algorithmes sont peu développés ou inexistants. Selon eux, le choix d'un numéro d'ordre de la racine dans la représentation symbolique d'un nombre algébrique n'est pas adapté. En effet , en adoptant la convention de Mathematica, quelles règles de transformation pourraient nous permettre de calculer $Root(X^2-2,2)\cdot Root(X^2-3,2)^2$ à partir de $Root(X^2-2,2)$ et $Root(X^2-3,2)$? Ils proposent une représentation alternative d'un nombre algébrique réel x, incluant le polynôme caractéristique P de degré n, et la suite des signes pris par les dérivées successives de P en x : $\{signe(P'(x)), signe(P''(x)),..., signe(P^{n-1}(x))\}$. Le lemme de René Thom assure que, pour deux racines réelles de P distinctes, les deux suites de signes sont distinctes. Cette représentation code donc les nombres algébriques réels de façon univoque. Les auteurs proposent des algorithmes basés sur ces codes pour la plupart des opérations arithmétiques. Ces algorithmes sont disponibles comme des ajouts au système Maple. D'autres travaux plus récents, s'intéressent également au calcul exact avec des nombres algébriques réels. (Rioboo 2003) résume les différentes représentations possibles et compare les performances des algorithmes associés, implantés dans le système Axiom.

Du point de vue de notre sujet, cet aperçu sur la représentation et la manipulation des nombres algébriques dans les systèmes de calcul formel nous permet déjà de tirer quelques enseignements :

- la distinction entre calcul numérique et calcul formel est peu pertinente. Des systèmes de calcul formel évaluent exactement des expressions ne comportant que des nombres, éventuellement algébriques. On préfèrera donc distinguer entre calcul exact et calcul approché ;
- la grande disparité des systèmes de calcul formel dans le traitement d'un aspect pourtant important -les nombres algébriques- montre à quel point la conception de ces outils est le fruit de choix entre plusieurs critères tels que la généralité, la performance, la convivialité, … Cette conception est également le reflet d'un niveau scientifique et technique tant dans le domaine informatique, que dans le domaine des mathématiques.
- perdre en généralité en se limitant à un domaine bien défini, tel que l'arithmétique sur les nombres algébriques réels par exemple, permet de bénéficier de techniques spécifiques efficaces.

1.1.4. Modélisation et simulation numérique dans le cadre d'un système de calcul formel

De la partie 1.1.1, il ressort que les systèmes de calcul formel permettent de manipuler des expressions mathématiques comprenant aussi bien des symboles que des nombres. La partie 1.1.2 a précisé que les systèmes de calcul formel pouvaient calculer de manière exacte avec les nombres entiers et les rationnels et en précision multiple avec les nombres réels. La partie 1.1.3 enfin a présenté la prise en charge partielle des nombres algébriques par les seuls systèmes de calcul formel. Il semble alors intéressant d'envisager ces environnements logiciels comme des outils privilégiés de la modélisation et de la simulation numérique sur ordinateurs.

Dans les parties suivantes, après avoir souligné l'intérêt de la formulation des modèles dans le cadre d'un système de calcul formel, nous détaillons trois approches consistant à mêler calcul numérique et calcul formel lors de la résolution. Une dernière partie précise les raisons qui font qu'aujourd'hui encore, la modélisation et la simulation numérique de systèmes industriels ne s'effectue pas au sein d'un système de calcul formel.

1.1.4.1. Formulation formelle des modèles

La modélisation dans le cadre d'un système de calcul formel, tout comme dans un environnement de calcul numérique comporte une étape de traduction des équations et inéquations algébriques et/ou différentielles régissant le système étudié, de la syntaxe mathématique usuelle vers la syntaxe de l'outil utilisé. Le choix des symboles représentant les opérateurs mathématiques usuels ainsi que les règles de précédence entre opérateurs étant presque consensuels, la conversion syntaxique consiste principalement à exprimer les noms des fonctions mathématiques usuelles telles que cosinus ou arc tangente à l'aide de la nomenclature retenue.

La question de la justesse sémantique de la traduction se pose également mais de façon apparemment moins cruciale que pour les environnements de calcul numérique. En effet, les systèmes de calcul formel proposent à la personne en charge de traduire informatiquement un modèle, un univers sémantique plus riche et plus proche des abstractions mathématiques. Comme on l'a vu précédemment, les nombres rationnels sont représentés de façon exacte par tous les systèmes de calcul formel, contrairement aux systèmes de calcul numérique. Les différences portent également sur les opérateurs logiques disponibles : aux opérateurs usuels $\neg$, $\wedge$, $\vee$, viennent s'ajouter $\Rightarrow$ et $\Leftrightarrow$. Certains environnements mettent de plus à disposition les quantificateurs logiques $\forall$ et $\exists$, de sorte que les modèles s'exprimant à l'aide de prédicats logiques sont aisément traduits dans l'outil de calcul formel. Tous les systèmes de calcul formel

intègrent les opérateurs fonctionnels de dérivation ou d'intégration, permettant de décrire des modèles fonctionnels, évitant ainsi l'utilisation d'une discrétisation préalable à la modélisation. Cette discrétisation –formule de différences finies, formule de quadrature, …- est toujours requise dans le cadre d'un environnement de calcul numérique où les fonctions ne peuvent être représentées en tant que telles. Axiom va très loin dans la richesse sémantique du modèle saisi puisqu'il permet de préciser la structure algébrique dans laquelle doit être recherchée la solution du problème posé.

La formulation formelle d'un modèle à l'aide d'un système de calcul formel a donc lieu la plupart du temps sans perte de sémantique et avec un minimum d'effort dans la conversion syntaxique.

1.1.4.2. Résolution formelle

Suite à la formulation formelle, la résolution proprement dite du modèle représenté peut être soit formelle, soit numérique. Une résolution formelle tente de produire la totalité des solutions exactes, exprimées sous la forme la plus explicite possible. Pour cela, toute la panoplie des algorithmes spécifiques du calcul formel est mise en œuvre, réalisant une suite de transformations à partir de l'expression initiale. Suivant la nature du modèle, la solution sera plus ou moins exploitable par l'utilisateur, soucieux de disposer de quantités numériques en fin de compte. La résolution formelle de l'équation algébrique $X^5 - X + 1 = 0$ présentée en 1.1.3.1 est un exemple de ce processus de résolution formelle, qui fournit toutes les racines de l'équation sous forme de nombres algébriques, définis implicitement, exacts certes mais peu exploitables d'un point de vue numérique. Toute résolution formelle se conclut donc par une approximation numérique de solutions formelles. Les modalités de l'évaluation numérique sont variées depuis l'isolement des racines, abordée notamment dans (Mourrain, Vrahatis, & Yakoubsohn 2001), jusqu'à l'approximation des racines par fractions continues, présentée dans (Brezinski 2004). Lors de cette étape interviennent évidemment bon nombre de méthodes du calcul numérique.

1.1.4.3. Résolution numérique améliorée par le calcul formel

A l'opposé de la résolution formelle, la résolution numérique dans un système de calcul formel s'attache à calculer une approximation numérique de tout ou partie des solutions en utilisant les algorithmes du calcul numérique. Les méthodes de résolution sont alors celles habituellement présentes dans les environnements de simulation numérique : Newton-Raphson pour les systèmes d'équations non linéaires (Dedieu 2006), Gear pour les systèmes d'équations algébro-différentielles (Gear 1971;Hall & Porsching 1990)… Dans ce cas, les techniques de

calcul formel, et en premier lieu la dérivation formelle, n'interviennent que pour calculer de manière exacte des expressions formelles requises par l'algorithme numérique de résolution. Des évaluations de ces expressions formelles, pour des valeurs numériques particulières de certains symboles, sont calculées chaque fois que l'algorithme l'exige. Mathematica se fait le chantre de cette approche nommée *« symbolically enhanced numerical computing »*. Maple adopte une approche similaire, bien que plus collaborative, en intégrant par exemple de manière transparente dans son environnement les routines de la bibliothèque mathématique Nag. Comme le souligne (Trefethen & Ford 2000), cette démarche ouvre à l'utilisateur la voie du développement d'algorithmes hybrides mêlant intimement calcul formel et calcul numérique.

1.1.4.4. Résolution par des méthodes analytiques approchées

Si la résolution formelle a tout son intérêt dans certains cas, et si la résolution numérique améliorée par le calcul formel permet la simulation de bon nombre de systèmes, depuis quelques années émerge une troisième approche, consistant à mettre en œuvre des méthodes analytiques approchées à l'aide de systèmes de calcul formel. L'un des adeptes de cette démarche, Rémi Barrère, note que *« le champ d'application du calcul formel s'est inutilement restreint aux méthodes algébriques exactes »*, alors que les méthodes analytiques approchées auraient leur place et un intérêt certain dans les systèmes de calcul formel. *« Les approximations symboliques combinent les avantages des calculs symboliques et des méthodes d'approximation. A l'opposé des méthodes numériques, elles permettent la manipulation d'expressions littérales avec des paramètres symboliques, conduisant ainsi à une meilleure compréhension, donc à une meilleure maîtrise des systèmes physiques sous-jacents. »* L'auteur transpose d'un point de vue algorithmique quelques méthodes classiques de l'analyse fonctionnelle. Sont abordés la méthode des approximations successives -basée sur le théorème du point fixe-, la méthode de Newton -dont le domaine d'application est les espaces de Banach, parmi lesquels figurent des espaces de fonctions-, les développements en série des perturbations et la méthode de Galerkin. Ces méthodes de l'analyse fonctionnelle sont implémentées à l'aide du langage de programmation fonctionnel et à base de règles proposé par Mathematica. La programmation fonctionnelle, où tout calcul est vu comme l'évaluation d'une fonction, paraît être particulièrement adaptée à la transposition des méthodes de l'analyse fonctionnelle.

A titre d'illustration, nous reprenons ci-après la présentation de la méthode des développements en série des perturbations faite par Rémi Barrère dans (Barrère 2005). L'objectif initial est la résolution d'un modèle légèrement perturbé $F(x, p+\varepsilon)=0$, connaissant la solution x_0 du modèle non perturbé $F(x,p)=0$. La solution du problème perturbé est recherchée sous la

forme d'une série entière tronquée $\sum_{k=0}^{l} \varepsilon^k x_k$, les inconnues étant la famille de coefficients $(x_k)_{k=1,\ldots,l}$. Ces coefficients sont obtenus par les identifications successives des développements de Taylor à l'ordre 1, 2, …, l au point 0 de $F(\sum_{k=0}^{l} \varepsilon^k x_k, p+\varepsilon)$ et de la fonction identiquement nulle. La méthode a été programmée dans Mathematica de manière récursive, en tirant parti de ses capacités de traitement des séries entières, des opérateurs fonctionnels du langage, et du calcul d'approximants de Padé (Brezinski 2004) afin d'améliorer la convergence de la série. La solution analytique approchée ainsi calculée permet, entre autres, de déduire aisément les sensibilités d'ordre 1, 2, …, l puisque $x(\varepsilon)$ est identifié à son développement de Taylor à l'ordre l au voisinage de 0 : $x(\varepsilon) = \sum_{k=0}^{l} \varepsilon^k \frac{x^{(k)}(0)}{k!}$. La méthode s'étend au cas d'un nombre quelconque de variables et de paramètres.

La démarche proposée est séduisante pour deux raisons :

1. sous réserve d'être dans le cadre d'application du théorème des fonctions implicites, elle s'applique à un opérateur F entre espaces de Banach, donc non seulement aux modèles régis par des équations algébriques, mais également à des systèmes intégraux, algébro-différentiels ou aux dérivées partielles ;
2. elle permet de disposer d'une solution du problème certes approchée, mais symbolique, donc toujours manipulable par le système de calcul formel.

En fait, les intérêts de cette méthode se retrouvent également dans les autres méthodes présentées par Rémi Barrère, car ils sont consubstantiels de l'analyse fonctionnelle et de sa transposition dans le cadre d'un système de calcul formel. Le cadre théorique établi de l'analyse fonctionnelle, et sa grande généralité, conduisent à des algorithmes très élégants en programmation fonctionnelle et à des résultats remarquables sur certains problèmes. Il ne faut cependant pas occulter certaines difficultés. Les méthodes proposées soulèvent des questions non triviales, telles que la divergence de séries entières, l'accélération de la convergence de telles séries, le calcul de dérivées de Fréchet, etc. De plus, contrairement aux apparences, la plupart de ces méthodes comprennent des étapes de calcul numérique, telles que la résolution de systèmes linéaires. Les résultats produits sont donc approchés, non seulement du fait de la nature de la méthode, mais également du fait de ces étapes de calcul numérique. Un écueil supplémentaire, et non des moindres, est la possible explosion de la taille des expressions symboliques manipulées avec l'augmentation de la taille du problème. Pour s'en convaincre, il n'est qu'à considérer la méthode des développements en série des perturbations dans le cas où le nombre d'inconnues, scalaires ou fonctions, est m et le nombre de paramètres est n. Dans ce cas, le développement

en série entière à l'ordre 0 comprend 1 vecteur de longueur m. Le développement en série entière à l'ordre l comprend $(l+1)^n$ vecteurs de longueur m[4].

1.1.4.5. Simulation numérique de systèmes industriels ?

L'expressivité des langages proposés par la plupart des systèmes de calcul formel, proches des notations mathématiques usuelles, jointe aux capacités de calcul formel, de calcul numérique, et aux riches possibilités graphiques, semble faire des systèmes de calcul formel des candidats sérieux à la simulation numérique. (Braun 2000) chante les louanges de la simulation utilisant le calcul formel, en énumérant les principales différences avec la simulation numérique :

1. il n'est pas nécessaire de donner une valeur numérique à chaque paramètre ;
2. une solution exacte est calculée, et non pas une approximation numérique ;
3. les modèles peuvent être paramétrés ;
4. les problèmes inverses sont faciles à résoudre ;
5. le calcul formel offre une grande souplesse ;
6. de nouvelles méthodes de résolution sont disponibles ;
7. l'optimisation devient possible car les expressions sont réduites ;
8. les outils proposent une démarche de travail intuitive et adaptée aux ingénieurs.

Exprimés ainsi ces avantages sont caricaturaux et certains erronés. Une solution exacte n'est bien évidemment pas produite pour tout problème et, quand elle l'est, elle peut être difficile à manipuler du fait de sa taille ou de sa forme. Si les problèmes inverses étaient faciles à résoudre, les géophysiciens notamment seraient ravis, qui obtiendraient aisément le profil du sous-sol à partir des sismogrammes ! Par contre, l'intérêt de manipuler des modèles paramétrés est soulignée, et ce point de vue mérite que l'on s'y attarde. L'auteur précise que *« la complexité croissante des caractéristiques des produits rend nécessaire une meilleure compréhension des paramètres des procédés »*. Selon lui, *« les méthodes numériques ne peuvent fournir de l'information relative à l'interaction et à l'interdépendance des paramètres. Elles sont efficaces uniquement si tous les paramètres physiques sont connus avec suffisamment de précision ; dans le cas contraire, l'efficacité est sensiblement réduite par la nécessité de calculer de nombreuses fois le même problème avec des paramètres variant légèrement. »* Selon lui, cette analyse de sensibilité aux paramètres du modèle serait donc plus aisée et plus efficace au sein d'un système

[4] Le nombre de coefficients d'un développement de Taylor à l'ordre l d'une fonction de n variables est égal à la dimension de l'espace $Q_l(\Re^n)$ des polynômes sur $\Re^n$, de degré maximal l en chacune des variables. $\dim Q_l(\Re^n) = (l+1)^n$

de calcul formel. Cette proposition semble exacte, la dérivation formelle d'expressions analytiques étant un des points forts du calcul formel. Dans le cas de modèles explicites, où les variables d'intérêt x_1, x_2, …, x_m s'expriment comme des fonctions explicites des paramètres p_1, p_2, …, p_n, les sensibilités $\partial^{(1,0,...,0)}x_1(p_1,p_2,...,p_n)$, $\partial^{0,1,...,0)}x_1(p_1,p_2,...,p_n)$, …, $\partial^{(0,0,...,1)}x_m(p_1,p_2,...,p_n)$ sont faciles à calculer formellement. Dans le cas -bien plus courant !- de modèles implicites, où des équations algébriques, différentielles ou intégrales relient variables et paramètres, sans que l'on puisse exprimer explicitement les premières en fonction des seconds, une expression analytique exacte de la fonction dérivée $(x_1^{'},x_2^{'},...,x_m^{'})$ est la plupart du temps inaccessible ! Comme on l'a vu au chapitre 1.1.4.4 la méthode analytique approchée, dite des développements en série des perturbations, comprend le calcul précis des sensibilités à un ordre quelconque, uniquement au point particulier $(p_1,p_2,...,p_n)$ pour lequel le modèle $F(x_1,...,x_m,p_1,...,p_n)=0$ a été résolu. Afin d'analyser la sensibilité des variables aux différents paramètres du modèle, il semble donc que nous soyons encore obligés, même dans le cadre du calcul formel, *« de calculer de nombreuses fois le même problème avec des paramètres variant légèrement. »* !

L'intégration systématique de méthodes analytiques approchées dans les nouveaux systèmes de calcul formel ouvrira peut être les portes de la simulation numérique de modèles complexes à ces outils. Pour l'instant, force est de constater que la littérature scientifique ne mentionne toujours pas la modélisation et la simulation de bout en bout d'un système industriel complet, décrit par au moins quelques dizaines de symboles –variables ou paramètres- et au moins quelques dizaines de relations. Les performances des algorithmes utilisés aujourd'hui sont pourtant devenues remarquables. A notre avis, la raison essentielle est le fait que les méthodes numériques apportent, dans le cadre complet d'environnements de résolution de problèmes, spécialisés ou généralistes, une réponse satisfaisante aux questions de la modélisation et de la simulation de systèmes ! Bien que souvent très riches, les systèmes de calcul formels actuels sont loin de couvrir toute l'étendue du processus de modélisation et de simulation. (Houstis & Rice 2000) décrit l'architecture logicielle d'un environnement de résolution de problèmes particulier -PELLPACK- spécialisé dans la résolution d'équations aux dérivées partielles. Les composants logiciels de l'environnement se répartissent en quatre catégories : ceux dédiés à la spécification du problème, ceux dédiés à la spécification de la solution, l'environnement d'exécution et enfin l'environnement de visualisation. Si l'on procède à une analyse critique des fonctionnalités des divers systèmes de calcul formel, au vu de cette classification, les manques sont criants :

1. spécification du problème : absence de bibliothèques de géométries, absence d'éditeurs de géométries, absence d'éditeurs de la logique discrète du système ;

2. spécification de la solution : absence de bases de connaissance ;
3. environnement d'exécution : faiblesse des interfaces de communication avec les environnements tiers ;
4. environnements de visualisation : absence ou faiblesse des outils d'analyse de performance, absence ou faiblesse de l'analyse de données.

L'interaction forte entre les différents éléments d'un système industriel[5] se traduit nécessairement par une interaction forte entre les différents éléments modélisant ce système. Aujourd'hui, les systèmes de calcul formels, monolithiques, ne savent pas inter-opérer avec d'autres composants logiciels pour fournir un environnement complet de modélisation et de simulation de systèmes industriels. Leur champ d'application privilégié reste encore les modèles continus dans le temps et dans l'espace, constitués d'un nombre réduit de symboles et de relations.

[5] En extrapolant les définitions fournies par la norme ISO 15 288 et (Association Française d'Ingénierie Système 2004), un système industriel peut être vu comme un ensemble d'éléments en interaction, organisés pour atteindre un ou plusieurs résultats déclarés, dans un contexte de production ou de transformation de biens et de services. Il est caractérisé par un ensemble de propriétés telles que fonctions assurées, interfaces externes, architectures, activités, ressources ... et est réalisé à l'aide d'un ensemble de constituants (matériels, logiciels, automates, opérateurs, processus ...). Il est rarement isolé et est en interaction avec son environnement (sociétal, économique, politique ...).

1.2. Calcul formel préprocesseur pour le calcul numérique

Dans le domaine de la simulation sur ordinateurs, il est à noter l'engouement pour une approche consistant à utiliser un système de calcul formel pour un prétraitement d'un modèle, afin d'obtenir des expressions mathématiques qui seront ensuite traitées de manière efficace dans un langage compilé. Afin de rendre compte des articles relatant l'utilisation d'un système de calcul formel préalablement à la simulation numérique, nous synthétisons la démarche globale adoptée par la plupart des auteurs. Une analyse précise des quatre étapes de cette démarche à travers plusieurs applications permet d'exhiber les fonctionnalités les plus utilisées des systèmes de calcul formel, lorsqu'ils sont des préprocesseurs pour le calcul numérique. Les bénéfices obtenus sont mis en regard des limites de l'approche.

1.2.1. Synthèse automatique de codes de simulation numérique à partir de systèmes de calcul formel

Comme souligné dans (Amberg, Tonhardt, & Winkler 1999), simuler un modèle *« implique le codage, soit d'un programme complètement nouveau pour l'application particulière, soit l'adaptation d'un logiciel commercial existant. »* Les inconvénients de chacune des options sont connus. *« Le temps et l'effort requis par l'écriture d'un code spécialisé sont souvent prohibitifs, et d'un autre côté il est difficile d'avoir le niveau de contrôle souvent nécessaire sur le modèle et la procédure de résolution, lorsqu'on utilise un logiciel commercial. »* Les auteurs notent une troisième possibilité qu'ils appliquent aux modèles régis par des équations aux dérivées partielles, mais dont le domaine de validité est bien évidemment beaucoup plus large : disposer d'une *« spécification de haut niveau du système* [aux dérivées partielles]*, à partir de laquelle un code de simulation est compilé. »* Il s'agit de spécifier non seulement le problème sous la forme d'équations, de condition initiales et de conditions aux limites, mais également de spécifier la procédure de résolution. Cette spécification doit se faire dans un cadre sémantique proche de celui de la personne amenée à modéliser. La fourniture automatique d'un code de simulation par traduction de la spécification est le but de cette démarche. Les auteurs en attendent les bénéfices suivants :

- un accès complet au modèle ;
- un accès à la partie numérique ;
- une efficacité dans le calcul numérique.

La spécification dans un langage de haut niveau du modèle à simuler doit être la transposition la plus complète et la plus fidèle possible du modèle pensé par la personne en

charge de la modélisation. La spécification dans un langage de haut niveau de la procédure de résolution doit permettre de s'affranchir de certaines étapes fastidieuses, telles que le processus de discrétisation, ou la transformation formelle de certaines expressions afin de les adapter à l'algorithme de calcul numérique. Le code de simulation produit automatiquement, et écrit dans un langage compilé, doit pouvoir s'ouvrir aux techniques du calcul numérique à hautes performances (parallélisme, bibliothèques mathématiques optimisées, etc.) pour justifier le processus de génération de code.

Le processus décrit précédemment peut être mis en œuvre de façon partielle. Tel est le cas lorsque le chercheur ou l'ingénieur utilise un système de calcul formel pour déterminer l'expression analytique d'une intégrale ou d'une dérivée, puis code lui-même cette expression dans un programme existant. De plus, il faut souligner que ce processus de modélisation n'est en aucun cas lié à l'utilisation d'un système de calcul formel. Modéliser un problème d'optimisation à l'aide de l'environnement GAMS (McCarl 2007), c'est également spécifier le modèle et la procédure de résolution dans un langage de haut niveau, puis produire automatiquement un code de simulation -en langage machine-. Modéliser un problème à partir d'équations aux dérivées partielles à l'aide de l'environnement FEMLAB (COMSOL AB. 2001), c'est également spécifier le modèle et la procédure de résolution dans un langage de haut niveau, puis produire automatiquement un code de simulation -en langage MATLAB-.

Le processus consistant à produire une *« spécification de haut niveau du système, à partir de laquelle un code de simulation est compilé»* n'exige donc pas l'utilisation de systèmes de calcul formel. Pourtant, ces derniers nous semblent pouvoir apporter des avantages certains lors de trois étapes de ce processus :

1. la spécification du problème à résoudre dans le langage proposé par le système de calcul formel est proche de la formulation initiale des équations. De plus, certains outils, Mathematica notamment, proposent la notion de fonction adaptée à la description fonctionnelle des modèles ;
2. la transformation d'expressions formelles, préalablement à la génération de code, est l'apanage des systèmes de calcul formel ;
3. la génération de code enfin, est une fonctionnalité de la plupart des systèmes de calcul formel. Les langages cibles, FORTRAN et C la plupart du temps, sont standardisés et non propriétaires. Le code produit est donc indépendant de l'outil qui l'a généré, et peut se prêter à toutes les manipulations par les environnements de développement du marché.

Afin de confirmer ou d'infirmer ces avantages, les chapitres suivants détaillent les choix adoptés par plusieurs auteurs lors des quatre étapes clés du processus : la spécification du problème, la spécification de la solution, la transformation formelle de la spécification et la synthèse de programme.

1.2.2. Spécification du problème

1.2.2.1. Caractéristiques communes aux spécifications de problèmes dans les systèmes de calcul formel

Il faut noter en premier lieu que les travaux utilisant un système de calcul formel pour la synthèse de programmes de calcul numérique s'appliquent, soit à un domaine particulier tel que la thermodynamique ou l'informatique financière, soit à une classe particulière de modèles, telle que les équations aux dérivées partielles. A notre connaissance, il n'existe pas d'environnement de résolution de problèmes multi-physiques (MPSE pour Multi-Purpose Problem Solving Environment) s'appuyant sur la génération automatique de code dans un système de calcul formel. Pour un panorama des environnements de résolution multi-physiques on lira notamment (Houstis & Rice 1995).

Le langage de la plupart des systèmes de calcul formel est soumis à une syntaxe voisine de celle adoptée couramment par les ingénieurs et les chercheurs pour écrire des expressions mathématiques. Il est donc naturel d'exprimer les équations et inéquations à résoudre dans le langage natif de l'outil. Cela est d'autant plus vrai que le modèle décrit subit déjà une traduction en un programme écrit dans un langage compilé. Afin d'éviter la complexité et les erreurs éventuelles liées à la manipulation de plusieurs langages, et à la mise en œuvre de plusieurs processus de traduction, les adeptes de cette démarche conservent le plus longtemps possible le même cadre sémantique.

La majorité des travaux scientifiques utilisant le calcul formel comme préprocesseur du calcul numérique, s'appuie sur un des deux systèmes de calcul formel les plus utilisés : Mathematica et Maple. Dans ce cas, les approches proposées donnent lieu à des programmes additionnels, écrits dans le langage de l'environnement choisi, et intégrés de façon transparente à ce dernier. Selon les auteurs et le système de calcul formel, ces programmes additionnels sont désignés par des termes variés : paquetages, modules, boîtes à outils, etc. Le but de ces programmes additionnels est de fournir une syntaxe et une sémantique enrichies pour formuler les problèmes à résoudre. Les dialectes ainsi construits conservent bien évidemment les caractéristiques des langages de base :

- la spécification au sein de Mathematica suivra les principes de la programmation à base de règles, ou de la programmation fonctionnelle, ou de la programmation procédurale, ou combinera divers aspects de ces modes de programmation ;
- la spécification au sein de Maple suivra les principes de la programmation procédurale.

Outre la qualité du langage de programmation offert par Mathematica, et à un degré moindre par Maple, les logiciels proposés tirent avantage de certaines caractéristiques rarement offertes dans les langages de programmation indépendants de tout environnement :

1. Une notation symbolique bidimensionnelle des équations, identique à celle rencontrée dans les textes mathématiques, rend la spécification claire ;
2. Les documents structurés comportant à la fois du texte de présentation et des expressions à évaluer –*« notebooks »* sous Mathematica- permettent de disposer d'une spécification complète, cohérente et accessible ;
3. La spécification du problème est construite par la programmation d'instructions, mais elle peut être facilitée par la présence d'une interface graphique programmée dans l'environnement de calcul formel ;
4. La richesse sémantique du langage proposé donne lieu à des spécifications concises.

Le point 2 est détaillé dans (Dall'Osso 2003). *« La documentation physique et mathématique est, par définition, à jour avec le code. De plus, la description du modèle dans le manuel utilisateur peut être extraite de la spécification en utilisant les fonctionnalités d'exportation des systèmes de calcul formel. La vérification du code par une personne extérieure est centrée sur le document de spécification. Cette vérification peut être faite par des personnes non adeptes des langages de programmation. »*

(Bunnin et al. 2000) illustre l'utilisation d'une interface graphique facilitant la saisie de la spécification du problème (point 3). L'interface intégrée dans le système Mathematica, propose une palette de commandes prédéfinies. Bien que rudimentaire, cette interface évite certaines erreurs liées à la saisie de commandes dans une syntaxe spécifique.

La spécification d'un système industriel en fonctionnement continu -un modèle de colonne de distillation réactive-, présentée dans (Alfradique & Castier 2005) confirme le point 4. Une vingtaine d'instructions Mathematica suffit pour définir le calcul des fractions molaires à partir du nombre de moles, écrire les lois de conservation, les équations d'équilibre chimique et demander la création du programme FORTRAN correspondant à la résolution du modèle. La spécification tire notamment parti de la possibilité de travailler avec des expressions symétriques, et des sommes et des produits d'un nombre quelconque de termes. Une autre facette de cette richesse sémantique des langages des systèmes de calcul formel, détaillée dans (Barrère 2005), a déjà été présentée : un symbole pouvant représenter non seulement un scalaire, mais aussi une fonction, les équations fonctionnelles s'expriment aisément.

A ces quatre points il convient d'ajouter les remarquables capacités de reconnaissance de motifs (« pattern matching ») propres à Mathematica, notamment exploitées par (Husa, Hinder,

& Lechner 2006) lors du développement de leur environnement de résolution d'équations tensorielles très complexes apparaissant par exemple en relativité numérique.

Au vu des remarques précédentes, il n'est donc pas étonnant que les développements logiciels s'appuyant sur des techniques de calcul formel sont presque toujours intégrés aux environnements Mathematica et Maple. Quelques exceptions existent cependant.

Le paquetage GENTRAN 90 (Borst, Goldman, & van Hulzen 1994) permettait la traduction d'expressions mathématiques écrites dans le système de calcul formel Reduce en FORTRAN 90. Ce produit n'est pratiquement plus utilisé.

RCMAG (Bastos & Monti 2005) accepte des spécifications de modèles mêlant des quantités numériques et symboliques, et synthétise des programmes compilés pour l'environnement de résolution de problèmes multi-physiques Virtual Test Bed (Brice, Gödkere, & Dougal 1998). La spécification doit suivre la syntaxe de description de circuits du langage PSPICE, à laquelle est rajoutée la possibilité d'utiliser des variables symboliques. Cette spécification du modèle est élaborée en suivant la méthode RCM (Resistive Companion Method) fondée sur le couplage naturel entre éléments d'un réseau. *« Le couplage naturel impose les lois physiques de conservation de l'énergie au niveau des ports du modèle »*, qu'il s'agisse d'un réseau électrique, ou d'un réseau constitué d'éléments non électriques.

Ctadel (van Engelen, Wolters, & Cats 1996) accepte des spécifications de problèmes décrits par des équations aux dérivées partielles. Le langage propriétaire comporte des mots-clés pour représenter des opérateurs tels que la dérivation partielle, l'intégration ou la divergence. Les équations sont constituées à partir de ces opérateurs, des opérateurs arithmétiques usuels et de symboles représentant les quantités variant dans le temps et l'espace. Les symboles ont ici une sémantique bien particulière. Contrairement aux systèmes de calcul formel, où les symboles sont vus comme des quantités continues, les symboles dans Ctadel sont évalués uniquement en certains points, constituant un domaine discrétisé de l'espace. La résolution des équations aux dérivées partielles par des méthodes de différences finies explique ce choix, dont une conséquence est le mélange de la spécification du problème avec la spécification de la solution.

1.2.2.2. Analyse de la spécification d'un modèle de diffusion dans les environnements SciNapse et femLego

Afin d'illustrer l'utilisation des deux systèmes de calcul formel pour la spécification de problèmes, nous nous intéressons plus particulièrement à l'environnement SciNapse (Akers et

al. 1998), qui s'intègre à Mathematica, et à la boîte à outils femLego (Amberg, Tonhardt, & Winkler 1999) qui s'intègre à Maple.

$$\exists E \subset \Re^2; \forall t \in \Re^+; \forall (x,y) \in E; D_1 f(t,x,y) = D_2 f(t,x,y) + D_3 f(t,x,y)$$

Équation 2 - Modèle de diffusion bidimensionnelle.

```
Region[0 <= x <= 2 && 0 <= y <= 3 && 0 <= t <= 1/2, Cartesian[{x,y},t]] ;
When[Interior, der[f, t] == der[f, {x,2}] + der[f, {y2}]; SolveFor[f]];
When[Boundary, f == 0];
When[InitialValue, f == 50 Sin[Pi x/2] Sin[Pi y/3]];
Movie[frames == 11];
RelativeErrorTolerance[.01];
Output[f, Labelled];
Dimensionless;
```

Figure 24 – Spécification d'un problème de diffusion bidimensionnelle sous SciNapse.

La spécification d'un problème de diffusion bidimensionnelle, décrit par Équation 2, à l'aide de SciNapse illustre l'utilisation des caractéristiques du système de calcul formel hôte Mathematica. La forme de la spécification, donnée à la Figure 24 pour un domaine d'étude rectangulaire et des conditions aux limites de type Dirichlet, décrit la géométrie, l'équation aux dérivées partielles, les conditions aux limites et la condition initiale, mais masque à l'utilisateur de SciNapse la réalisation algorithmique. Les structures de contrôle, telles que les ruptures de séquence ou les boucles, sont absentes de cette spécification, la rendant totalement déclarative, donc a priori simple et manipulable. Quelques mots-clés (**Region**, **When**, …) ont été adjoints au langage du système de calcul formel. Les actions sémantiques associées aux expressions correspondantes ont été définies, par les concepteurs de SciNapse, soit aux moyens de règles de transformation, soit au moyen de procédures. Ce choix de réalisation est caché à l'utilisateur, et n'est d'ailleurs pas détaillé dans l'article de référence.

Specify PDEs

```
[ > # Diffusion equation with gradient BC;
> veq:=ElementInt(phi(x,y)*(v(x,y) - vo(x,y))/dt)=
        -Diffusivity* ElementInt( nab(phi(x,y))[i] &t nab(v(x,y))[i] )
         + BoundaryInt(bigQ*phi(x,y)*qBCval) ;
```

$$veq := \text{ElementInt}\left(\frac{\phi(x,y)\,(\mathrm{v}(x,y)-\mathrm{vo}(x,y))}{dt}\right)= -\mathit{Diffusivity}\ \text{ElementInt}\left(\left(\frac{\partial}{\partial x}\phi(x,y)\right)\left(\frac{\partial}{\partial x}\mathrm{v}(x,y)\right)+\left(\frac{\partial}{\partial y}\phi(x,y)\right)\left(\frac{\partial}{\partial y}\mathrm{v}(x,y)\right)\right) + \text{BoundaryInt}\,(\mathit{bigQ}\ \phi(x,y)\ \mathit{qBCval})$$

```
>
  # qBCval is a predefined name. It is used as a boundary value parameter
  which is specified with the mesh. It can thus take different values on
  different parts of the boundary. In this example it can be used to set
  the influx to different values on different parts of the boundary.;
```

Define input lists

```
[ > # set up lists that are used repeatedly as input:
[ > eq_list:= [veq]:
[ > unknown_list:=  [v(x,y)]:
[ > old_unknown_list:=  [vo(x,y)]:
[ > params_list:= [Diffusivity,bigQ]:
```

Dirichlet boundary conditions

```
[ > # Dirichlet BCs;
[ > DBCueq:=   v(x,y)=x:
> mkDirBC([DBCueq],unknown_list,old_unknown_list,
  params_list,`2DP1_gsq_bsf`):
```

adddbc

SPdirb

Specify initial conditions

```
[ > # Initial conditions. ic_eq is on the form 'unknown = expr':
> # variables that do not enter in lhs of an ic_eq are given the intial
  value 0:
> mkIC([ v(x,y)=x*x+y*y ],  unknown_list, params_list , `2DP1_gsq_bsf`);
```

icmkre

icamdp

icadr1

icadm1

3

Figure 25 – Spécification d'un problème de diffusion bidimensionnelle sous femLego.

Les auteurs de femLego illustrent l'utilisation de leur boîte à outils avec le calcul d'un problème de diffusion bidimensionnelle similaire au cas traité par SciNapse. Les conditions aux limites sont légèrement différentes : une condition de Dirichlet est imposée sur une partie S_1 de

la frontière, et une condition de Neumann est imposée sur une autre partie S_2 de la frontière. La spécification du problème reprise à la Figure 25 prend pourtant une forme radicalement différente de celle fournie pour SciNapse. L'équation de diffusion n'est pas écrite dans sa forme originale mais sous une forme variationnelle, après application d'une formule aux différences finies pour la dérivée en temps. La spécification du modèle n'est absolument pas générale, c'est-à-dire indépendante de la méthode et des outils choisis pour la résolution, et ce pour au moins quatre raisons :

1. La projection de l'équation de diffusion sur un espace Φ de fonctions tests est lié au choix de la méthode des éléments finis pour résoudre le problème ;
2. L'application d'une formule aux différences finies pour la dérivée en temps est un choix de résolution et non de formulation du problème : une formulation éléments finis en temps et en espace serait également envisageable ;
3. L'intégrale le long de la surface S_2 est en fait calculée le long de toute la frontière en faisant intervenir un paramètre prédéfini `qBCval`, valant 1 en tout point de S_2, et 0 ailleurs. Les valeurs du paramètre `qBCval` sont affectées lors de la définition du maillage éléments finis de la structure modélisée à l'aide d'un logiciel de maillage externe.
4. Les conditions aux limites de Dirichlet et de Neumann sont exprimées différemment : la condition aux limites de Neumann est intégrée à la formulation variationnelle tandis que la condition de Dirichlet est exprimée séparément.

Les auteurs justifient leur préférence à *«travailler avec la forme variationnelle, au lieu d'essayer d'automatiser son obtention à partir des équations aux dérivées partielles. La raison de cela est qu'il est plus facile de travailler directement avec la forme variationnelle lorsque l'on veut contrôler finement la partie numérique, par exemple rendre implicite un terme particulier dans les équations, etc. »* L'argumentaire explique donc le mélange entre la spécification du problème et la spécification de sa solution par la nécessité de manipuler un modèle transformé, plus proche de celui traité par les algorithmes numériques de résolution. La séparation quasi complète des spécifications dans l'environnement SciNapse, permettant tout de même le traitement d'un modèle similaire, nous semble pourtant souhaitable. Dans cet environnement, seule la distinction des variables et des paramètres faite au moment de la définition des équations du modèle, déroge à la règle. A l'opposé, femLego précise la liste des variables et des paramètres après la définition des équations du modèle ! La réconciliation des deux objectifs apparemment antagonistes -la spécification du problème indépendamment de la spécification de la solution, et le choix précis des expressions mathématiques manipulées par les algorithmes de résolution numérique- est une des justifications du travail présenté.

1.2.2.3. Limites des spécifications de problèmes dans les systèmes de calcul formel

Le chapitre précédent a mis en lumière deux choix de conception contestables dans des outils de spécification de modèles, intégrés à un système de calcul formel : l'utilisation d'une formulation transformée de l'équation originale dans femLego, et le mélange des spécifications dans SciNapse. Ces choix sont le fait des concepteurs des boîtes à outils respectives.

A l'inverse, certaines limites dans la spécification de modèles de systèmes industriels complexes ne résultent pas de mauvais choix de conception, mais sont structurellement liées à leur réalisation au sein de systèmes de calcul formel. Les systèmes industriels complexes sont le plus souvent des systèmes hybrides, composites, et multi-physiques. Leur spécification s'accommode donc bien, voire requiert, trois fonctionnalités :

1. un formalisme de représentation de modèles discontinus ;
2. des primitives de couplage de modèles ;
3. des bibliothèques de modèles préétablis et réutilisables.

Les points suivants analysent la faiblesse ou l'absence de réponses des systèmes de calcul formel à ces trois besoins.

1.2.2.3.1. Absence d'un formalisme de représentation de modèles discontinus

Comme le note (Faure, Davenport, & Naciri 2000), si *« la force principale du calcul formel est l'obtention de familles d'expressions en un seul calcul* [… dans lesquelles certains symboles] *font office de paramètres »*, dans les systèmes de calcul formel courants aucune détection d'incohérence liée à des valeurs particulières des paramètres n'est effectuée. De plus, aucun domaine de variation particulier n'étant assigné à un symbole, et l'évaluation de chaque expression produisant une expression unique le calcul symbolique, tel qu'il est habituellement mis en œuvre, n'est en rien une réponse à la représentation des modèles discontinus. (Faure, Davenport, & Naciri 2000) introduit des expressions multi-valuées particulières, appelées expressions conditionnelles, qui semblent être une avancée par rapport à la structure de contrôle conditionnelle **`If`** telle qu'elle est traitée dans Mathematica. Dans une expression conditionnelle, chaque *« valeur potentielle est associée à une condition sur des paramètres »*. La syntaxe de telles expressions conditionnelles est la suivante :

```
IFF((condition1, valeur1), (condition2, valeur2), …, (conditionn, valeurn))
```

Indépendamment de son objectif initial -éviter l'incohérence de certains calculs dans lesquels interviennent des paramètres- cette introduction de la notion de fonction par morceaux semble intéressante pour représenter des modèles discontinus. Si elle permet d'envisager sérieusement la représentation d'événements de temps entraînant des changements de modèles, elle est hélas incapable de prendre en compte des événements d'état car *« il est supposé que seuls des paramètres apparaissent dans les conditions »*. De ce point de vue, la construction syntaxique **`When[région, équations]`** introduite dans SciNapse est supérieure. En effet une région, ou domaine spatio-temporel, y est définie par une condition logique portant sur l'ensemble des symboles, variables ou paramètres, du modèle. Les discontinuités spatiales et temporelles du modèle sont donc décrites dans un formalisme unique et élégant. Mais cette formulation n'est en aucun cas un modèle du comportement hybride du système, manipulable en tant que tel à des fins d'analyse de fonctionnement, de contrôle, etc. Aucun des systèmes de calcul formel largement diffusé ne propose de formalisme discret, que ce soit un formalisme de machines à états, ou un formalisme de type réseau de Petri, et à notre connaissance la littérature ne relate pas d'expérience de couplage entre un simulateur de systèmes discrets et un système de calcul formel.

1.2.2.3.2. Absence de primitives de couplage de modèles

L'approche usuelle en modélisation de systèmes (continus) est la construction d'un modèle complet à partir de modèles élémentaires des différentes parties du système à représenter. Le processus de composition de modèles peut prendre diverses formes. La plus courante est la mise en relation de deux modèles par l'identification de symboles propres à chacun de ces modèles. La sémantique de cette identification est claire : les valeurs numériques des symboles associés deux à deux sont égales. L'environnement de modélisation et de simulation MATLAB/SIMULINK est construit sur ce principe. Il en est de même du langage de modélisation Modelica (Fritzson & Engelson 1998). En pratique, la construction du modèle complet s'appuie sur une interface graphique permettant de choisir des modèles élémentaires, représentés par icônes spécifiques, et de les relier par des arcs symbolisant les connexions. La simplicité et la généralité du concept font que ce type d'interface d'accès aux modèles et à leurs relations est intégrée à bon nombre de systèmes de simulation numérique. Son absence des systèmes de calcul formel actuels peut laisser penser que ces produits ne visent pas la modélisation et la simulation de modèles complexes. L'unique solution de composition de modèles disponible dans les systèmes de calcul formel est la concaténation des listes d'équations des différents modèles dans une liste unique. Cela est illustré dans (Alfradique & Castier 2005) où un bilan enthalpique, un bilan massique, une équation d'équilibre entre phases et une équation

d'équilibre chimique sont concaténés afin de constituer l'ensemble des équations d'un étage de la colonne à distiller réactive. Ce modèle est ensuite concaténé au modèle dérivé pour constituer un modèle complet d'un étage, intégrant les sensibilités analytiques. Concaténer ainsi des équations ou des modèles impose que les éléments combinés suivent une nomenclature unique, à moins bien sûr d'effectuer des remplacements préalables de certains symboles par les symboles de la nomenclature. Ce respect d'une nomenclature unique assure une cohérence entre les différents éléments du modèle spécifié. S'il est souhaitable dans le cadre de la modélisation par une seule et même personne, il est illusoire et contre-productif dans le cas d'un modèle complexe dont chaque élément est spécifié par une ou plusieurs personnes. Dans ce cas l'assemblage de modèles repose le plus souvent sur deux principes complémentaires :

1. chaque modèle d'un système particulier est vu comme une instance nommée d'un modèle générique représentant tous les systèmes régis par des lois identiques ;
2. des paires de symboles intervenant chacun dans un modèle distinct sont identifiées comme représentant la même grandeur.

La programmation orientée objet trouve naturellement sa place dans l'application informatique du principe 1 : une classe est le modèle informatique structurel de tous les systèmes régis par des lois identiques, tandis qu'un objet est le modèle structurel et comportemental d'un système particulier. Le langage à objets Modelica met en œuvre ce principe en proposant de regrouper dans une classe les équations caractéristiques d'un système, les attributs de la classe étant notamment les symboles intervenant dans ces équations.

Le principe 2 est le plus souvent appliqué en introduisant dans le langage de spécification du problème la possibilité de connecter entre elles des grandeurs issues de modèles distincts. Les équations de connexion proposées par Modelica peuvent être vues comme une généralisation de l'identification de paires de symboles : deux symboles peuvent bien sûr être considérées comme identiques mais, outre l'égalité, d'autres lois de connexion (lois de Kirchoff) peuvent être établies entre plusieurs symboles suivant la nature physique de la grandeur qu'ils représentent (Mattsson, Elmqvist, & Otter 1998).

1.2.2.3.3. Absence de bibliothèques de modèles préétablis et réutilisables

Un dernier obstacle à la spécification de systèmes industriels complexes à l'aide de systèmes de calcul formel est l'impossibilité de constituer des bibliothèques de modèles à échanger et à réutiliser. Les systèmes de calcul formel les plus utilisés se sont enrichis de boîtes à outils, couvrant les traitements spécifiques à certaines branches des mathématiques ou des sciences physiques. Mathematica propose notamment des compléments dédiés au contrôle, au traitement d'images ou à l'analyse de données expérimentales. De même, Maple peut intégrer

des boîtes à outils relatives à la géométrie différentielle, à la résistance des matériaux, ou à la logique floue. Toutefois, la capitalisation de modèles dans ces environnements est restée embryonnaire. Une première raison, et à notre avis la plus importante, découle du point précédent : l'impossibilité de combiner des modèles au travers d'une interface graphique freine la réutilisation des modèles. (Barker & Zhuang 1997) supputaient que *« l'application* [du calcul formel] *à l'ingénierie du contrôle avait été assez limitée, sans doute car les premiers logiciels n'étaient pas particulièrement conviviaux »*. Leur article relate l'interfaçage des environnements SIMULINK et Mathematica, aboutissant à un environnement dans lequel *« les modèles sont construits au moyen d'une interface graphique compréhensible et bien développée, familière à l'ingénieur de contrôle, et les résultats analytiques obtenus avec Mathematica peuvent être comparés directement avec les résultats de simulation obtenus avec SIMULINK »*. Les autres raisons de l'absence de capitalisation sur les modèles dans les systèmes de calcul formel sont à rechercher dans les faiblesses de ceux-ci par rapport aux langages et aux environnements de modélisation récents. Les concepteurs du langage de modélisation Modelica (Elmqvist, Mattsson, & Otter 1999) insistent sur le fait que *« la réutilisation de modèles est un point clé pour prendre en charge la complexité »*. *« Le but de Modelica est de servir de format standard de sorte que les modèles provenant de différents domaines puissent être échangés entre les outils et entre les utilisateurs »*. Pour atteindre cet objectif de réutilisation des modèles, plusieurs moyens sont mis en œuvre, techniques d'une part et organisationnels d'autre part. L'effort de conception du langage est guidé par deux concepts : la conception et la programmation orientées objet, et la modélisation acausale. La puissance de ces concepts, alliée aux technologies éprouvées issues de langages de modélisation existants, justifie pour les auteurs *« une nouvelle tentative d'introduire l'interopérabilité et l'ouverture dans le monde des systèmes de modélisation et de simulation »*. Outre les équations différentielles ordinaires ou les équations algébro-différentielles caractérisant des systèmes continus, Modelica prend en charge plusieurs formalismes ouvrant la porte à la simulation de systèmes hybrides complexes : les *« bond graphs »*, les machines à états finis, les réseaux de Petri, etc. Du point de vue organisationnel, une démarche de standardisation, avec la constitution d'un groupe d'experts délivrant une première spécification du langage, et l'existence d'une association à but non lucratif chargée du cycle de vie de ce dernier, démontrent une volonté de promouvoir le langage, les outils de simulation le mettant en œuvre et les bibliothèques de modèles. Une bibliothèque des éléments de modélisation les plus couramment utilisés, Modelica Standard Library, est maintenue par l'association afin de favoriser l'échange de modèles construits sur une base commune. A l'opposé des plates-formes comme Modelica, les systèmes de calcul formel pâtissent de trois lacunes, obstacles à la constitution et à l'échange de bibliothèques de modèles :

1. l'absence de formalismes variés, notamment pour spécifier les systèmes discrets ;
2. l'absence d'éléments structurants au dessus de la notion d'expression mathématique, alors que les langages de modélisation orientés objet structurent les équations spécifiques d'un élément de modélisation dans une classe, et classifient les modèles via les notions d'héritage et de composition des classes ;
3. l'absence d'interopérabilité avec les langages de modélisation largement utilisés comme VHDL-AMS, Verilog-AMS, MAST ou Modelica.

1.2.2.4. Des systèmes inadaptés à la spécification de modèles hybrides complexes

Des chapitres précédents, il découle que les systèmes de calcul formel actuels restent inadaptés pour spécifier des modèles hybrides complexes. L'absence de formalisme discret et la difficulté à réutiliser des modèles en les combinant, imposent de se détourner de démarches où la spécification du modèle, sa transformation, et la synthèse d'un programme de simulation ont toutes pour cadre un système de calcul formel. La spécification du problème a tout intérêt à être exprimée dans un des langages de modélisation orientés objet et équations, largement utilisés, tel que Modelica, ACSL, VHDL-AMS, Verilog-AMS, MAST ou gPROMS. De plus, comme le remarquent (Chernukhin et al. 2005) *« la complexité de la conception de systèmes réels implique aujourd'hui des personnes de divers horizons scientifiques. Il est très improbable que chacune de ces disciplines aura appris à utiliser les mêmes langages ou le même simulateur »*. La spécification d'un modèle complexe est donc vouée à être exprimée, non pas dans un des langages de modélisation précédents, mais bien dans plusieurs de ces langages. Cela conduit les auteurs de (Chernukhin, Polenov, Chandrasekhar, Solodovnik, Mantooth, & Dougal 2005) au choix d'un format de représentation de l'information sur le modèle qui soit indépendant des langages de modélisation. Les auteurs ont défini un schéma XML particulier décrivant la syntaxe dans laquelle doivent être exprimées les modèles. Les équations sont écrites dans une application XML standardisée, MathML (Ausbrooks et al. 2003), dédiée à la représentation des expressions mathématiques. Les traductions dans chacun des langages de modélisation usuels sont le fait d'un outil, MultiTranslator, utilisant comme référence un modèle exprimé dans le dialecte XML. Cette approche pragmatique qui s'applique à relever le challenge de l'interopérabilité lors de la spécification, et également lors de la simulation a des adeptes. ModelicaXML (Pop et al. 2005), une représentation XML du langage Modelica, est un autre candidat au titre de dialecte XML de référence pour la spécification de modèles. Quel que soit le dialecte retenu, au dire de (Pop, Savga, Aßmann, & Fritzson 2005) *« les techniques de composition et de transformation de logiciels peuvent s'appliquer aux langages de modélisation et de simulation de systèmes physiques utilisés aujourd'hui »* afin d'encadrer la composition et la transformation de composants de modélisation. Un des objectifs des auteurs est la *« transformation symbolique*

d'équations ». C'est donc sans doute en relation avec le dialecte XML de référence pour la spécification de modèles, et plus particulièrement avec l'application MathML, que se doit d'intervenir le calcul formel dans le processus de modélisation et de simulation de systèmes complexes.

1.2.3. Spécification de la solution

Par opposition à la spécification du problème, la spécification de la solution détaille le système amené à le résoudre. Le système amené à résoudre le problème étant informatique, cette spécification couvre principalement deux domaines : l'algorithmique et l'architecture informatique, logicielle ou matérielle. Tout particulièrement lors de la modélisation et de la simulation de systèmes décrits par des équations, la spécification du problème et la spécification de la solution sont très liées puisque les méthodes, les outils et les contraintes de résolution sont guidées, voire dictées par la spécification du problème. Cependant, comme affirmé dans (Zualkernan & Tsai 1988), *« l'interdépendance de la spécification du problème et de la spécification du système* [pour le résoudre] *ne signifie pas qu'elles ne sont pas des types de description différentes et qu'elles ne devraient pas être construites explicitement et séparément »*. En pratique, dans le cas de l'utilisation de systèmes de calcul formel pour la synthèse de programmes de simulation numérique, la spécification du problème, composée essentiellement des équations, inéquations et du formalisme discret représentant le modèle, devrait être séparée de la spécification de la solution, composée notamment du langage de programmation cible, de la méthode de résolution, des critères de performance sur la solution et des fournitures attendues. En effet, d'une part les informations manipulées dans les deux spécifications sont de nature différentes et auront peut être intérêt à être exprimées dans des langages différents. D'autre part, si la modélisation d'un élément évolue peu, la simulation de celui-ci peut prendre des formes multiples selon l'architecture informatique cible, les méthodes de résolution choisies, les critères de performance attendus ou la représentation des résultats souhaitée. Cette séparation entre les deux types de spécification est monnaie courante dans les langages de modélisation orientés objets et équations. A aucun moment dans la spécification du langage Modelica (Modelica Association 2005) n'apparaît un quelconque élément de spécification des solutions.

L'étude des articles détaillant des processus où le système de calcul formel est utilisé pour la synthèse de programmes de simulation numérique, montre que :

1. la spécification du problème et la spécification de la solution ne sont jamais séparées ;
2. les deux spécifications sont exprimées dans le même langage ;

3. le langage de spécification adopté est le langage de programmation proposé par le système de calcul formel.

Si spécifier le problème dans le langage de programmation proposé semble tout à fait cohérent lorsqu'il s'agit de décrire un système continu par des équations et des inéquations, la spécification d'un système hybride se heurte à l'absence de formalisme discret déjà évoqué. Le choix de spécifier la solution dans le langage natif du système de calcul formel semble plutôt un choix par défaut, voire un non choix ! Pire encore, la plupart du temps la spécification du problème et la spécification de la solution sont entremêlées, nuisant ainsi à la lisibilité du modèle et anéantissant tout espoir de réutilisation des modèles.

Un effort est fait dans (Amberg, Tonhardt, & Winkler 1999) pour d'abord spécifier le problème par la fourniture de l'équation de diffusion, des conditions aux limites et des conditions initiales, puis ensuite spécifier la solution par la nature du graphique à produire, les solveurs de systèmes linéaires utilisés, le type d'élément fini, etc. Mais, comme cela a été détaillé en 1.2.2.2, des éléments de spécification de la solution apparaissent lors de la spécification du problème.

Au vu du cas d'étude présenté en 1.2.2.2, l'environnement SciNapse (Akers, Baffes, Kant, Randall, Steinberg, & Young 1998), semble proposer un cadre syntaxique plus cohérent permettant une séparation des deux spécifications presque complète. En fait, la structure **When** proposée pour définir l'ensemble des équations valides pour un domaine spatio-temporel, peut inclure non seulement les symboles choisis comme étant les variables du problème, mais également le type de discrétisation en espace ou en temps du domaine. Ainsi, dans (Randall & Kant 1997) les auteurs spécifient, dans une seule et même instruction **When**, à la fois le modèle - équation de Black-Scholes représentant l'évolution d'une option sur un marché financier- et la discrétisation de celui-ci en imposant aux discrétisations par différences finies du temps et de la variable d'espace de respecter le critère de stabilité de Cranck-Nicholson.

La spécification des systèmes régis par des équations aux dérivées partielles, telle que présentée dans (van Engelen, Wolters, & Cats 1996), mêle les équations et la description du schéma aux différences finies adopté pour la résolution. La spécification peut également inclure le mode de présentation des équations lors de la génération d'un rapport au format LaTeX.

(Korelc 2001) présente un système de génération automatique de procédures numériques destinées à différents environnements de simulation par éléments finis. La génération de code traduit une seule et même description symbolique du problème. Toutefois la spécification du modèle ne ressemble en rien à la formulation des équations sur le papier : la méthode de Galerkin exploite une formulation intégrale de l'équation de conduction, et les dérivées partielles

présentes dans le modèle sont formulées comme le résultat d'un processus de différentation automatique.

Les quelques exemples précités suffisent à illustrer le fait que lors de l'utilisation d'un système de calcul formel pour synthétiser un programme de simulation numérique, des éléments de la spécification de la solution viennent systématiquement polluer la spécification du problème. A côté de choix de modélisation apparaissent des choix de résolution. Par analogie avec le développement d'architectures informatiques, à côté de choix de conception apparaissent des choix d'implémentation. Or les choix d'implémentation, ici de résolution, sont voués à évoluer différemment -plus vite la plupart du temps- des choix de conception, ici de modélisation. Le génie logiciel nous enseigne à séparer ceux-ci, et à les mettre en relation par un niveau intermédiaire afin de concilier les cycles de vie différents des deux domaines : le modèle de conception « pont » référencé dans (Gamma et al. 1999) est une solution possible. Dans notre cas, l'absence de séparation entre la spécification du problème et la spécification de la solution annihile tout espoir de réutilisation des modèles. A notre avis, elle découle de deux raisons :

1. dans des environnements dédiés à une classe de modèles ou à une classe de méthodes de résolution, il peut paraître inutile d'imposer cette séparation vue comme une contrainte supplémentaire imposée à la fois au développeur de l'environnement et à ses utilisateurs ;
2. les systèmes de calcul formel sont inadaptés à la mise en œuvre de cette séparation par application du modèle de conception « pont ». La programmation orientée objets nous semble être un pré-requis afin de classifier les choix de résolution selon une hiérarchie de classes, et de relier la résolution du modèle à certains choix de résolution uniquement au moment de la simulation.

1.2.4. Transformation formelle de la spécification

Avant de procéder à la génération du code de simulation numérique, les environnements évoqués effectuent diverses étapes de manipulations formelles et éventuellement des étapes d'approximations numériques selon le principe introduit en 1.1.1.2. La description de ces étapes constitue le code de la boîte à outil complétant le système de calcul formel. Cette description s'exprime dans le langage du système de calcul formel. Rien d'étonnant alors à ce que, selon le langage de programmation proposé -Maple ou Mathematica la plupart du temps- la forme de la spécification soit différente. Sous Maple, les étapes de transformation sont exprimées par des fonctions. La spécification est alors composée d'une suite d'appels à certaines de ces fonctions. Sous Mathematica, les étapes de transformation sont plutôt exprimées par des règles de remplacement tirant partie de la reconnaissance de motifs offerte. La spécification est alors composée d'une suite de déclarations, le système se chargeant d'appliquer successivement les règles disponibles pour produire les expressions qui donneront lieu à la génération de code. Cela explique d'ailleurs que SciNapse, construit au-dessus de Mathematica, mette à disposition un

langage de spécification du problème plutôt déclaratif, alors que femLego, construit au-dessus de Maple, laisse transparaître son langage procédural dans le langage de spécification du problème qu'il propose.

Les étapes d'approximation numérique, outre les opérations arithmétiques évidemment, sont essentiellement les évaluations numériques d'intégrales lors de l'application de la méthode des éléments finis pour résoudre des équations aux dérivées partielles. Les intégrales sont calculées formellement après évaluation numérique partielle de l'intégrande -un produit scalaire-, ou approximées par une formule de quadrature.

Les étapes de transformation formelle agissent sur la spécification du problème, c'est-à-dire les équations du modèle, pour produire, selon la spécification de la solution, un système d'équations adapté à une résolution numérique.

(Dall'Osso 2003) détaille les différentes transformations formelles utilisées lors de la synthèse d'un code de simulation numérique d'un système régi par une équation différentielle du second ordre. Après la spécification du problème, et avant la traduction dans le langage cible, l'auteur identifie les cinq étapes suivantes :

1. *« choisir une méthode de discrétisation et définir les règles de transformation qui permettront de traduire une équation différentielle en un système d'équations algébriques ;*
2. *appliquer les règles de transformation et obtenir un système linéaire ;*
3. *déterminer la structure de la matrice et l'écrire sous une forme adaptée à une résolution facile ;*
4. *organiser les relations donnant les coefficients de la matrice en ordre séquentiel et écrire l'algorithme de résolution ;*
5. *définir une organisation modulaire du code ».*

Ce processus peut être affiné et surtout généralisé à d'autres types de problèmes et à d'autres méthodes de résolution que celle mise en œuvre dans le cas traité. L'étape 1 consiste en fait à écrire la spécification de la solution, par le codage de certaines règles de transformation permettant de passer de la formulation continue du modèle à une formulation discrétisée. Le résultat de l'étape 2 est le système d'équations discrétisé. Dans l'étape 3 sont choisies les structures de données en fonction de la méthode de résolution. L'étape 4 regroupe l'optimisation des expressions générées avant la synthèse de programme et les choix algorithmiques. L'étape 5 relève plus de la conception logicielle de l'environnement de synthèse de programmes, que de la transformation automatique d'expressions.

Bien que cohérente, la description par (Dall'Osso 2003) du processus de conversion d'équations (aux dérivées partielles) d'une forme continue vers une forme discrète reste

informelle. (Akers, Baffes, Kant, Randall, Steinberg, & Young 1998) formalise ce processus en définissant des niveaux de transformation et de choix successifs, mis en œuvre dans la boîte à outils SciNapse. Les trois niveaux supérieurs sont **CoordinateFree**, **Component** et **Discrete**. Certaines transformations formelles permettent le passage du niveau **CoordinateFree** au niveau **Component**. D'autres transformations formelles permettent le passage du niveau **Component** au niveau **Discrete**. Plus précisément, *« si les équations aux dérivées partielles sont données sous une forme indépendante du système de coordonnées, c'est-à-dire en terme de divergence, gradient et rotationnel, ces équations doivent être transformées en équations constitutives utilisant le système de coordonnées physique. Les équations constitutives sont encore continues, elles doivent donc être mises sous une forme discrétisée, adaptée au traitement sur ordinateur. L'utilisateur peut spécifier des équations à n'importe quel niveau : indépendant du système de coordonnées, constitutif, ou discret, mais afin d'éviter des détails inutiles et de rendre la maintenance et l'optimisation faciles, les équations et les expressions de sortie devraient être exprimées au niveau le plus haut dans la spécification. »* (Randall & Kant 1997) note l'intérêt de spécifier un modèle par une équation vectorielle au niveau **CoordinateFree,** équation qui sera remplacée automatiquement par *« un ensemble d'équations scalaires dans un système de coordonnées particulier »* au niveau **Component**.

Elégante et efficace la décomposition en niveaux adoptée par les concepteurs du système SciNapse demeure assez spécifique du domaine d'application de l'outil : la résolution des équations aux dérivées partielles par une méthode de différences finies. D'une part, la méthode des éléments finis contrairement à une méthode de différences finies, requiert la manipulation continuelle de systèmes de coordonnées différents. La boîte à outils femLego (Amberg, Tonhardt, & Winkler 1999) propose deux fonctions, **loc2glob_def** et **glob2loc_def**, pour procéder aux changements entre un système de coordonnées local à un élément géométrique et le système de coordonnées global. Ces changements de systèmes de coordonnées intervenant successivement en cours de calcul, il ne s'agirait pas de travailler à des niveaux hiérarchiques différents -**LocalCoordinate** et **GlobalCoordinate** par exemple-, mais bien d'alterner des contextes de calcul **LocalCoordinate** et **GlobalCoordinate**. Par ailleurs, les transformations formelles menant du niveau **Component** au niveau **Discrete** prennent des formes variées selon la nature mathématique du modèle à simuler.

Au vu de cette brève analyse, il nous semble que les étapes de manipulations formelles associées à l'interprétation de la spécification par l'outil appartiennent toujours à l'une des trois catégories suivantes :

1. des transformations symboliques pour produire un système d'équations sous une forme adaptée à une résolution par une méthode numérique. Il s'agit du processus de discrétisation

inhérent aux familles de méthodes numériques les plus utilisées : les différences finies et les éléments finis ;

2. des simplifications algébriques des équations obtenues ;
3. des transformations symboliques pour produire, à partir des équations simplifiées un système d'équations équivalent, mais meilleur du point de vue de certains critères de performance sur le programme synthétisé.

Les trois chapitres suivants détaillent chacune de ces catégories de transformations formelles, successivement appliquées aux équations constituant la spécification. Un quatrième chapitre s'intéresse aux transformations symboliques portant non plus sur les équations, mais sur les structures de contrôle constituant l'algorithme de résolution, et dont le but est encore l'optimisation du programme synthétisé.

1.2.4.1. Discrétisation du système d'équations initial

Lorsque le modèle est un système d'équations non linéaires, et la méthode de résolution choisie est Newton-Raphson, ces transformations incluent le calcul des résidus par différence entre les membre droits et gauches des équations, et la dérivation formelle de ces résidus afin d'obtenir l'opérateur de Newton.

Lorsque le modèle est un système d'équations aux dérivées partielles, et la méthode de résolution choisie est l'application d'un schéma aux différences finies répondant au critère de stabilité de Cranck-Nicholson (Akers, Baffes, Kant, Randall, Steinberg, & Young 1998), ces transformations comprennent le remplacement formel des dérivées par les formules aux différence finies dans les équations initiales, la génération de l'ensemble des équations discrétisées à partir du système générique précédent, des conditions initiales et des conditions aux limites et enfin la mise sous une forme canonique du système d'équations linéaires, ou non linéaires, produit.

Lorsque le modèle est un système d'équations aux dérivées partielles, et la méthode des éléments finis est choisie (Amberg, Tonhardt, & Winkler 1999), ces transformations comprennent la dérivation formelle des fonctions de base, l'intégration formelle de produits scalaires de fonctions de base, la sommation de termes intégraux, et enfin la mise sous une forme canonique du système d'équations linéaires, ou non linéaires, produit.

1.2.4.2. Simplification algébrique des équations

Le noyau de calcul formel GPAS –General Purpose Algebraic Simplifier- illustre l'effort particulier de simplification algébrique des équations dans l'outil de génération de code Ctadel (van Engelen, Wolters, & Cats 1997). GPAS simplifie les expressions issues de la

discrétisation par différences finies d'équations aux dérivées partielles. Contrairement aux autres systèmes, souvent construits autour de règles explicites de réécriture des expressions, ce système de calcul formel s'appuie sur les propriétés des opérateurs, propriétés modélisées sous la forme d'une hiérarchie orientée objet de classes. De nouveaux opérateurs possédant des propriétés particulières peuvent être insérés dans cette hiérarchie de façon simple. *« Par exemple, de nouveaux opérateurs FFT peuvent être déclarés comme des instances de la classe opérateur linéaire et de la classe opérateur auto-commutatif. »*

1.2.4.3. Optimisation des équations

Le choix de la synthèse de programmes de simulation numérique écrits dans un langage compilé, à partir d'un système de calcul formel est d'abord guidé par le souci de la performance. (Castier 1999) avance que les programmes modernes de calcul formel *« fournissent un environnement de calcul complet pour l'implantation de modèles thermodynamiques.* […] *Cependant, les calculs numériques dans les systèmes de calcul formel peuvent être lents comparés aux codes visant le même objectif dans des langages tels que FORTRAN 77, FORTRAN 90 ou C. »* Par contre, il est possible et souhaitable de tout mettre en œuvre, dès la transformation de la spécification dans le système de calcul formel, pour obtenir un code synthétisé performant du point de vue du temps de calcul.

```
x=(-15*u**4)/(1 - u**2)**3.5 - (18*u**2)/(1 - u**2)**2.5 - 3/(1
- u**2)**1.5

v=u**2
w=1 - v
x=(-15*v**2)/w**3.5 - (18*v)/w**2.5 - 3/w**1.5
```

Figure 26 – Partage des sous-expressions communes lors du calcul d'une expression mathématique.

La recherche de la performance du code synthétisé conduit l'auteur de la boîte à outils Thermath à détecter les expressions communes à plusieurs expressions mathématiques afin de ne les calculer qu'une fois. Une sous-expression commune **C**, présente dans plusieurs expressions est d'abord affectée à une variable intermédiaire **I**. Toutes les occurrences de **C** sont alors remplacées par le symbole **I**, évitant ainsi dans le code synthétisé de multiples calculs de **C**. La Figure 26 illustre ce mécanisme d'optimisation de code lors de l'évaluation de l'expression mathématique $-\frac{15u^4}{(1-u^2)^{7/2}}-\frac{18u^2}{(1-u^2)^{5/2}}-\frac{3}{(1-u^2)^{1/2}}$. La transcription littérale en FORTRAN de cette expression algébrique est avantageusement remplacée par une séquence de trois instructions. Les variables intermédiaires v et w stockent les résultats d'évaluations partielles.

Ces résultats sont ensuite réutilisés lors de l'évaluation de l'expression initiale, dans laquelle les variables intermédiaires ont été substitués aux sous-expressions communes.

L'élimination des sous-expressions communes est une technique d'optimisation classique dans la démarche présentée ici. Les systèmes de calcul formel intègrent toutes les fonctionnalités requises pour cela : la reconnaissance de motifs et des primitives de remplacement. Les gains obtenus peuvent être considérables dès lors que sont évaluées des dérivées partielles. (Wolfe 1982) remarque que, alors que le temps de calcul d'une fonction et de ses n dérivées partielles peut être estimé à $n+1$ fois le temps de calcul de la fonction si le code de calcul transpose naïvement les expressions mathématiques, le rapport obtenu peut être réduit à 1,5 et rarement à plus de 2. Ce rapport constaté est d'ailleurs confirmé par les résultats théoriques présentés dans (Baur & Strassen 1983). Le processus d'élimination des sous-expressions communes est accessible dans Maple au travers du paquetage codegen fourni avec le système de calcul formel. La boîte à outils femLego (Amberg, Tonhardt, & Winkler 1999) tire partie de cette fonctionnalité intégrée pour *« identifier des éléments qui sont égaux dans des matrices symétriques, et éviter des évaluations inutiles »*. Ctadel va encore plus loin dans la reconnaissance de sous-expressions communes. *« Alors que GPAS optimise à une échelle locale, DICE –Domain-shift Invariant Common subexpression Eliminator- supprime des sous-expressions à une échelle globale. »* DICE reconnaît un très grand nombre de types de sous-expressions communes entre deux expressions données. Pour cela l'outil s'appuie sur les propriétés spécifiques des opérateurs, telles que l'associativité et la commutativité, mais aussi sur la substitution d'indices formels dans des expressions indicées. A titre d'exemple, si l'expression E_1 vaut $\sum_{j=c}^{d}\sum_{i=a}^{b} u_{i,j}$ et si l'expression E_2 vaut $\sum_{j=c}^{d} u_{i,j}$, DICE reconnaît la dépendance $E_1 = \sum_{i=a}^{b} E_2$. Si l'expression E_1 vaut $v_{i+a} + u_{i+a}$ et si l'expression E_2 vaut $u_i + v_i$, DICE reconnaît que E_1 est identique à E_2 dans laquelle i a été remplacée par $i+a$. Le processus de collecte de sous-expressions communes basé sur de telles règles s'avère particulièrement efficace dans le cadre de la discrétisation d'équations aux dérivées partielles ou d'équations différentielles intégrales où des formules de différences finies et/ou des formules de quadrature sont appliquées. L'analyse des sous-expressions communes à l'échelle globale du modèle discrétisé conduit à disposer d'une séquence de calcul où les évaluations multiples d'une seule et même expression sont quasiment supprimées. En contre partie, le stockage de l'évaluation de ces sous-expressions communes dans des variables intermédiaires engendre un surcoût lié aux opérations d'accès à la mémoire. Le système Ctadel permet de trouver un compromis entre le coût des opérations d'accès à la mémoire et le coût des opérations arithmétiques, compromis dépendant de l'architecture informatique qui accueillera la simulation numérique. Suivant le coût relatif des

différentes opérations sur une architecture matérielle cible, un algorithme itératif décide d'extraire ou non une sous-expression commune pour l'évaluer préalablement.

La reconnaissance des sous-expressions communes est une technique d'optimisation implantée dans de nombreux compilateurs. On pourrait donc penser que ce type d'optimisation devrait plutôt prendre place lors de la compilation du code synthétisé. Toutefois, (Dall'Osso 2006) confirme que *« l'élimination de sous-expressions communes exécutée par un système de calcul formel est plus efficace que celle exécutée par les compilateurs. Le traitement des expressions réalisé par un système de calcul formel n'est pas un simple remplacement basé sur la reconnaissance de motifs. Lors de la recherche de sous-expressions communes celui-ci prend en compte des règles mathématiques. »* Le système Ctadel illustre parfaitement ce processus d'optimisation des équations basé sur des règles mathématiques.

1.2.4.4. Optimisation des structures de contrôle

(Dall'Osso 2006) remarque que, outre l'élimination de sous-expressions communes, *« d'autres sortes d'optimisations que les compilateurs effectuent, telles que […], le déroulement de boucles, la permutation de blocs Do-If, la mise en ligne ou la réplication de code, peuvent être spécifiées par le développeur à l'aide d'un système de calcul formel »*. Il s'agit dès lors d'appliquer des transformations formelles, non plus aux équations, mais aux structures de contrôle réalisant l'algorithme de résolution choisi. Ce processus de transformation de la spécification de la solution s'inscrit naturellement dans le cadre d'un système de calcul formel proposant un langage fonctionnel où toutes les phrases du langage, expressions arithmétiques ou structures de contrôles, sont des fonctions. Il n'est donc pas étonnant que l'auteur propose une boîte à outils -MathCompile- à intégrer dans le système Mathematica pour pratiquer les différentes optimisations proposées. L'objectif visé est de donner au développeur le contrôle du processus d'optimisation lors de la phase de spécification de l'algorithme. Contrairement à une optimisation par un compilateur, *« le développeur décide quelle transformation doit être faite et à quel endroit »*. Les concepteurs de la boîte à outils SciNapse exploitent également le caractère fonctionnel du langage Mathematica. Après la construction des expressions constituant le programme au niveau hiérarchique **`Program`**, les structures de contrôle sont optimisées au niveau hiérarchique **`Code`**, dernier niveau avant la synthèse de programme proprement dite.

L'utilisateur de la boîte à outils MathCompile détaille la totalité des transformations formelles qui, à partir de la spécification du problème et de la spécification de la solution, conduiront au code en entrée de la synthèse de programme. Les fonctions de MathCompile lui permettent de procéder à une élimination de sous-expressions communes dans certaines

équations, ou à une permutation de blocs **Do-If** dans une portion de code. A l'opposé, l'utilisateur de la boîte à outils SciNapse fournit uniquement une spécification du problème et une spécification de la solution. Les transformations formelles, les optimisations notamment, lui sont inaccessibles. La première démarche vise un contrôle fin des transformations par un développeur, tandis que la seconde démarche a d'abord le soucis de faciliter la formalisation d'un modèle à simuler par un utilisateur.

1.2.5. Génération de code

La synthèse de code de simulation numérique est une fonctionnalité accessible à partir de systèmes de calcul formel depuis longtemps. Le générateur de code GENTRAN pour les systèmes de calcul formel Reduce et Macsyma est présenté dans (Gates 1985). Les premières versions du système de calcul formel Macsyma intégraient déjà la traduction d'expressions mathématiques manipulées en FORTRAN ou en C. En 1987, le générateur de code FORTRAN MACROFORT (Chancelier, Gomez, & Quadrat 1987), issu des travaux de l'INRIA, complète les fonctionnalités de ce système en permettant la génération de procédures complètes et non pas seulement la traduction d'expressions. Ce générateur est porté sur le système de calcul formel Maple peu après (Gomez 1990) et est encore disponible aujourd'hui comme boîte à outils complémentaire des dernières versions de Maple. Les principes de fonctionnement de MACROFORT sont rappelés dans (Gomez & Scott 1998). *«* [La synthèse de code] *s'appuie sur la procédure Maple fortran qui traduit des expressions algébriques, des matrices ou des séquences d'expressions Maple dans la syntaxe FORTRAN. Macrofort fournit un moyen de générer un programme indépendant, une fonction ou une procédure dans sa totalité incluant les déclarations de types, les structures telles que par exemple l'allocation de mémoire pour les tableaux, etc., et tout cela à l'intérieur de l'environnement Maple. »* L'utilisateur décrit chaque instruction élémentaire, ou chaque structure de contrôle, à l'aide d'une liste Maple. MACROFORT traduit chacune de ces listes en une ou plusieurs instructions FORTRAN. Un des points forts de l'outil est la possibilité de vectoriser efficacement le code synthétisé en procédant à des analyses de dépendances fines et à des restructurations, et donc de l'optimiser pour les architectures vectorielles. Dans le même article, les auteurs présentent la boîte à outils Transfor pour Maple. Construite au-dessus de MACROFORT, cette dernière traduit des programmes Maple complets en programmes FORTRAN indépendants. Sa spécificité réside dans la recherche de performance du code synthétisé : les procédures BLAS sont systématiquement utilisées lors de la traduction en FORTRAN d'opérations matricielles écrites en Maple.

Complété par les boîtes à outils MACROFORT et Transfor, ou à travers les fonctions du paquetage intégré codegen, Maple dispose sans doute des fonctionnalités de synthèse de code FORTRAN et C les plus abouties. Le système de calcul formel MuPAD permet lui aussi de produire des routines complètes. A l'inverse, Mathematica exprime uniquement une expression donnée dans différentes syntaxes -FORTRAN, C ou MathML-, mais ignore la notion de déclaration de types, de données ou d'entête de routine. La boîte à outils Thermath (Castier 1999) produit une procédure FORTRAN complète d'évaluation à partir d'une liste d'expressions arithmétiques à la syntaxe Mathematica, après une élimination des sous-expressions communes. La boîte à outils MathCompile (Dall'Osso 2003) se veut plus générale puisque le code Mathematica à traduire ne se réduit pas à une liste d'expressions arithmétiques, mais peut comporter des structures de contrôle.

Si les différents systèmes de calcul formel incluent tous par défaut la traduction d'expressions dans la syntaxe d'un langage compilé, FORTRAN ou C, la synthèse de codes complets de simulation numérique est le fait de boîtes à outils tierces le plus souvent issues de la recherche. Des résultats intéressants sont obtenus du point de vue du temps de calcul avec les codes synthétisés en adaptant la traduction à l'architecture matérielle et logicielle cible. Tel est le cas de Transfor. Un autre exemple remarquable est la possibilité pour le système Ctadel, à partir d'une spécification du problème unique, de synthétiser un code de simulation numérique adapté soit à une architecture séquentielle, soit à une architecture vectorielle, soit à une architecture à mémoire partagée ou encore à une architecture à mémoire distribuée (van Engelen, Wolters, & Cats 1996). La synthèse automatique de codes va ainsi dans le sens d'une réutilisation générative (*« generative reuse »*) des modèles, un mode de réutilisation détaillé dans (Mili et al. 2002).

La simulation numérique utilise le code synthétisé par le système de calcul formel. Ce dernier, parce qu'il est le fruit d'un processus de traduction automatique, et parce qu'il fait apparaître des variables temporaires liées à l'étape d'élimination des sous-expressions communes, n'est pas toujours d'une grande lisibilité. Les codes FORTRAN synthétisés par la boîte à outils Thermath (Castier 1999), pour le calcul de modèles thermodynamiques par équations d'état, en sont un témoignage. (Dall'Osso 2003) réfute la critique du manque de lisibilité et de la difficulté de maintenance des programmes générés automatiquement en avançant que le véritable code source ce sont les commandes écrites dans le langage du système de calcul formel. Si cet argument peut répondre au besoin de lisibilité de la spécification, il n'apporte rien à la question de l'interopérabilité du code synthétisé avec les autres logiciels constituant l'environnement de simulation numérique : logiciels de maillage, d'analyse de données, etc.

L'approche consistant à utiliser les systèmes de calcul formel comme des pré-processeurs de la simulation numérique est pertinente, puisqu'elle a donné lieu à des réalisations remarquables. Toutefois, le fait qu'elle occulte la nécessité de l'interopérabilité la restreint à des solutions de simulation numérique très spécifiques et souvent propriétaires. Les produits SciNapse et Ctadel, l'un spécialisé dans l'informatique financière et l'autre intégré dans le système de simulation météorologique Hirlam, en sont des exemples. Malgré le cadre de spécification très général offert par le calcul formel, les boîtes à outils intégrées à ces systèmes se spécialisent dans une famille de modèles et/ou une famille de méthodes numériques. Pire encore, la réutilisation par composition (*« compositional reuse »*), au centre des langages de modélisation orientés équations tels que Modelica, n'est pas abordée. Les raisons en sont évidemment multiples. Les langages de programmation des systèmes de calcul formel n'intègrent pas les notions de conception et de programmation par objets utiles pour organiser les données et les algorithmes. La réutilisation par composition suppose de convenir d'interfaces standardisées pour les spécifications des problèmes et les spécifications des solutions afin de pouvoir combiner les unes et les autres. Enfin, la réutilisation par composition impose une grande fiabilité des composants logiciels unitaires amenés à collaborer.

1.3. Calcul formel à l'intérieur d'un environnement de calcul numérique ?

Comme cela a été vu en 1.1.1.2, les systèmes de calcul formel évaluent une expression mathématique en procédant à une succession d'étapes de calcul formel et/ou de calcul numérique. Pour cela, outre des algorithmes spécifiques du calcul formel, ces systèmes incorporent les algorithmes classiques du calcul numérique pour la résolution de systèmes d'équations algébriques ou différentielles, l'intégration numérique, etc. Dès lors, il est naturel de penser à inverser le paradigme de la simulation numérique à l'intérieur d'un système de calcul formel : pourquoi ne pas faire du calcul formel à l'intérieur d'un environnement de calcul numérique ? Cela semble d'autant plus cohérent que les besoins en calcul « exact » dans un processus de simulation numérique semblent bien identifiés : il s'agit la plupart du temps d'évaluer une expression analytique où interviennent par exemple des dérivations partielles, des intégrations ou des séries entières. Les parties 1.3.1 et 1.3.2 envisagent des situations où le calcul formel semble être intégré au calcul numérique, mais ne l'est pas. La synthèse du Chapitre 1, en partie 1.4, précise en quoi consisterait le fait de faire vraiment du *« calcul formel à l'intérieur d'un environnement de calcul numérique »*.

1.3.1. Génération d'une expression à la syntaxe du langage compilé

Lorsque la ou les expressions analytiques à déterminer ne sont pas amenées à varier au fil des simulations numériques, la solution adoptée est simple et efficace. Un système de calcul formel est utilisé pour évaluer ces expressions, et le résultat de cette évaluation formelle est inséré dans le ou les logiciels de simulation numérique. La fonctionnalité de génération de code du système de calcul formel est souvent utilisée pour traduire l'expression évaluée dans la syntaxe du langage compilé. Cette démarche est adoptée par (Taylor 1997) pour déterminer l'expression formelle de propriétés thermodynamiques à l'aide de Maple. L'auteur obtient celles-ci pour un nombre arbitraire de constituants dans le mélange, ce qui confère aux expressions formelles un caractère générique donc réutilisable. Dans ce cas, la simulation numérique utilise des expressions arithmétiques produites par le calcul formel, mais n'incorpore aucune fonctionnalité de calcul formel. Une fois intégrées au code de simulation, les expressions obtenues sont figées et ne peuvent plus être manipulées par le programme comme c'était le cas dans le système de calcul formel. Seule est possible leur évaluation numérique, après affectation d'une valeur numérique à toutes les variables de l'expression. Cette évaluation numérique est soumise aux aléas de l'arithmétique réelle approchée rappelés en 1.1.2.2.

1.3.2. Différentiation automatique

Lorsque l'expression à transformer n'est pas uniquement algébrique, c'est-à dire constituée à partir des opérateurs arithmétiques et des fonctions usuelles, mais incorpore des structures de contrôle, celle-ci constitue un programme. Comme on l'a vu en 1.2.4.4, l'approche fonctionnelle du calcul formel permet de transformer de telles expressions, par exemple pour optimiser le temps de calcul du code synthétisé. Le programme transformé est traduit dans un langage compilé, automatiquement via une boîte à outils complémentaire du système de calcul formel, ou manuellement.

Une des transformations usuelles d'un programme, calculant la valeur $F(x_1, x_2, ..., x_m)$ d'une fonction F de $\Re^m$ dans $\Re^n$ en un point particulier $(x_1, x_2, ..., x_m)$, est celle qui consiste à produire un nouveau programme calculant à la fois $F(x_1, x_2, ..., x_m)$ et $F'(x_1, x_2, ..., x_m)$. Ce nouveau programme sera par exemple utilisé pour résoudre le problème « trouver $(x_1, x_2, ..., x_m) \in \Re^m$ tel que $F(x_1, x_2, ..., x_m) = 0$ » à l'aide d'une méthode de gradient. Ainsi dans Thermath (Castier 1999), boîte à outils complémentaire de Mathematica, l'auteur construit-il la liste des résidus associés à des équations, puis concatène à cette liste les dérivées des résidus par rapport à chacune des variables. La liste obtenue est ensuite traduite en un programme FORTRAN.

Lorsque le programme calculant F est déjà disponible dans un langage compilé tel que FORTRAN 77, FORTRAN 90, C ou C++, le programme calculant $F(x_1, x_2, ..., x_m)$ et $F'(x_1, x_2, ..., x_m)$ n'est pas obtenu à l'aide d'un système de calcul formel, mais plutôt par différentiation automatique. Après avoir introduit le principe de la différentiation automatique, nous précisons les deux variantes pour la mise en œuvre de ce principe.

1.3.2.1. Principe de la différentiation automatique

(Mischler et al.) présente le principe fondamental de la différentiation automatique. *« Une séquence d'instructions peut être considérée comme la composition des fonctions représentant le calcul réalisé par chaque instruction. Dès lors, la technique est basée sur un ensemble de règles de différentiation des opérations élémentaires du langage considéré et sur la règle dite de chaînage pour la différentiation des fonctions composées. »*

Considérons la fonction f, définie de $\Re^m$ dans $\Re^p$, et la fonction g, définie de $\Re^n$ dans $\Re^m$. En désignant par $D_x f$ la différentielle totale de la fonction f calculée au point x, la règle de chaînage s'écrit :

$$D_x(f \circ g) = (D_{g(x)} f) \circ (D_x g)$$

Elle permet de calculer la différentielle de la fonction associée à la séquence d'instructions $\{g(x), f(g(x))\}$ en procédant à l'évaluation de la séquence d'instructions $\{D_x g, g(x), (D_{g(x)} f) \circ (D_x g), f(g(x))\}$.

La transposition de la règle de chaînage précédente ${}^t D_x(f \circ g) = ({}^t D_x g) \circ ({}^t D_{g(x)} f)$ conduit au mode inverse de différentiation automatique, par opposition au mode direct. Le mode direct est plutôt employé lorsque $p > n$, le mode inverse lorsque $p < n$.

L'application du principe de différentiation automatique conduit à développer des analyseurs de code couplés à des outils de génération de code. La phase de génération de code ajoute les déclarations des variables différentielles aux déclarations des variables originales et les instructions de calcul des différentielles aux instructions de calcul des fonctions. Le programme initial est augmenté de données et d'opérations supplémentaires. (Fagan, Hascoet, & Utke 2006) précise que l'augmentation des variables dans le programme transformé peut être réalisée selon deux approches principales :

- la séparation complète des variables originales et des variables supplémentaires ;
- l'encapsulation des variables originales et des variables supplémentaires.

1.3.2.2. Séparation complète des variables originales et des variables différentielles

L'approche de séparation complète des variables originales et des variables différentielles consiste à conserver telles quelles toutes les déclarations originales des variables, et à déclarer séparément les nouvelles variables différentielles. De façon analogue, les instructions originales sont conservées, et des instructions de calcul des différentielles sont ajoutées. La Figure 27 illustre cette approche : au texte FORTRAN 77 initial constituant le code de la fonction f sont adjoints des déclarations et des instructions supplémentaires pour le calcul de $f'(x)$. Ces déclarations et instructions supplémentaires apparaissent en rouge.

```
subroutine af(x,y,ax,ay)
      real au
      common/aS/ au
      real u,p
      common/S/ u,p
      real ax,ay,at
      real x,y,t
      at=2*x*ax
      t=x**2
      au=au+at
      u=u+t
      ay=2*(ax*u+x*au)
      y=2*x*u+p
end subroutine
```

Figure 27 – Différentiation automatique par séparation complète des variables.

Cette approche est celle adoptée par les premiers logiciels de différentiation automatique, notamment par le plus connu : ADIFOR (Bischof et al. 1992), mais aussi par l'environnement Odyssée (Faure 2005) issu de l'INRIA.

1.3.2.3. Encapsulation complète des variables originales et des variables différentielles

```
#include <math.h>
class C {
private:
struct S {
      aFloat u;
      float p;} s;
public:
void f(aFloat x, aFloat y){
      aFloat t;
      t = pow(x,2);
      s.u = s.u+t;
      y=2*x*s.u+s.p;}
};
```

Figure 28 – Différentiation automatique par encapsulation complète des variables.

A l'opposé de la séparation complète, l'encapsulation complète des variables originales et des variables différentielles associe chaque variable différentielle introduite avec la variable originale correspondante. Les variables appariées sont déclarées à l'aide d'un nouveau type qui encapsule les deux données. En ce qui concerne les traitements, chaque instruction doit porter non seulement sur la variable originale mais aussi sur la variable différentielle. Selon (Fagan, Hascoet, & Utke 2006), *« le choix technique évident est d'encapsuler les opérations augmentées*

et originales dans de nouvelles opérations (surchargées) ». Cette approche a été rendue possible par FORTRAN 90 et C++ principalement, offrant à la fois des types de données structurées et la surcharge des opérateurs du langage. La Figure 28 présente le texte C++ correspondant au calcul de la fonction f précédente. Les opérateurs et la fonction surchargés apparaissent en rouge, le type de données encapsulé en vert.

L'élégance de cette approche de la différentiation automatique qui requiert uniquement l'utilisation d'une bibliothèque définissant les types encapsulés et les opérations surchargées, et apporte des modifications restreintes au code initial, a fait le succès d'Adol-C (Griewank, Juedes, & Utke 1996), différentiateur automatique de codes C et C++. Par ailleurs, des bibliothèques de différentiation efficaces à intégrer manuellement à des programmes de calcul voient le jour. (Straka 2005) fournit à des codes FORTRAN 90 des dérivées numériques d'ordre un, à la précision machine. Sur le même principe, (von Hippel 2006) calcule des dérivées numériques pour des ordres de dérivation éventuellement supérieurs à deux. La bibliothèque C++ FastDer++ (Tijskens et al. 2004) combine des techniques de différentiation automatique et de vectorisation pour calculer efficacement des dérivées numériques précises lors de la résolution d'équations aux dérivées partielles.

1.4. Synthèse: calcul formel, calcul numérique, calcul symbolico-numérique

Au fil des chapitres précédents, émerge une classification des démarches adoptées pour traiter la modélisation et la simulation de systèmes régis par des équations mathématiques. Comme le résume le tableau ci-dessous, chaque démarche représente, transforme et évalue les modèles différemment.

<table>
<tr><th colspan="2"></th><th>Calcul formel</th><th>Différentiation automatique</th><th>Calcul numérique</th></tr>
<tr><td colspan="2">Modèles représentés</td><td>Expressions symbolico-numériques</td><td>Expressions numériques augmentées d'une information numérique sur leurs dérivées</td><td>Expressions numériques</td></tr>
<tr><td colspan="2">Transformation des modèles en cours d'exécution</td><td>Oui</td><td colspan="2">Non</td></tr>
<tr><td rowspan="5">Modes d'évaluation des modèles</td><td>Evaluation symbolico-numérique</td><td>Oui</td><td colspan="2">Non</td></tr>
<tr><td>Arithmétique réelle en précison machine</td><td>Oui</td><td colspan="2">Oui</td></tr>
<tr><td>Arithmétique réelle à précison multiple</td><td>Oui</td><td colspan="2">Oui
(bibliothèques mathématiques)</td></tr>
<tr><td>Arithmétique réelle à intervalles en précision multiple</td><td>Oui
(boîtes à outils)</td><td colspan="2">Oui
(bibliothèques mathématiques)</td></tr>
<tr><td>« Arithmétique réelle exacte »</td><td>Non</td><td colspan="2">Oui
(bibliothèques mathématiques)</td></tr>
</table>

Tableau 1 – Représentation, transformation et évaluation des modèles en calcul formel et en calcul numérique.

Le calcul formel représente des modèles où figurent des symboles et des nombres, en précision machine ou en précision arbitraire. Les modèles du calcul numérique ne comportent que des nombres, de même que les modèles issus de la différentiation automatique. Ces derniers sont enrichis de données complémentaires, mais qui restent des quantités numériques.

Dans un système de calcul formel, les expressions mathématiques sont systématiquement représentées par des structures de données dynamiques. Lors de l'exécution d'un programme

écrit dans le langage de ce système, ces expressions subissent des transformations formelles et des approximations numériques. Ce paradigme est poussé à l'extrême dans les systèmes de calcul formel proposant un langage de programmation fonctionnel car, dans ce cas, l'exécution d'une instruction quelconque, ou du programme complet, consiste à appliquer une fonction à des arguments, et donc à produire une nouvelle expression. A l'opposé, les expressions mathématiques du calcul numérique, représentées par des séquences d'instructions d'évaluation numérique, constituent des modèles statiques. Les codes issus de la différentiation automatique en sont un cas particulier : même s'ils comportent plus d'expressions que le code initial, ces expressions ne sont pas transformées en cours d'exécution, mais seulement évaluées numériquement.

L'évaluation d'une expression dans un système de calcul formel produit soit une expression mêlant symboles et nombres, soit un nombre réel ou complexe issu d'un calcul à précision multiple (cf. 1.1.2), soit un nombre réel ou complexe en précision machine. Des boîtes à outils complémentaires rajoutent l'arithmétique réelle à intervalles en précision multiple (cf. 1.1.2.4) aux systèmes de calcul formel. La plupart du temps, l'évaluation d'une expression dans un logiciel de calcul numérique produit un nombre réel ou complexe en précision machine. Des bibliothèques spécifiques permettent cependant de disposer d'une arithmétique réelle à précision multiple, d'une arithmétique réelle à intervalles en précision multiple, voire d'une arithmétique réelle exacte (cf. 1.1.2.4).

Faire du calcul formel exige de représenter les expressions mathématiques sous la forme de structures de données auxquelles des transformations puissent être appliquées en cours d'exécution du programme. Faire du calcul formel à partir d'un environnement de simulation numérique exige donc la définition, dans le langage compilé utilisé, des types de données et des opérations élémentaires qui permettent ensuite l'écriture d'algorithmes de transformation formelle. Aujourd'hui, les systèmes de calcul formel ne mettent pas directement à disposition du développeur logiciel FORTRAN ou C++ les types de données et les opérations élémentaires qu'ils définissent pour leurs besoins. L'interface proposée est généralement un protocole de communication acceptant une expression mathématique et retournant le résultat de son évaluation.

L'interaction entre calcul numérique et calcul formel telle qu'elle est aujourd'hui dictée par les systèmes de calcul formel est-elle satisfaisante ? La simulation numérique au sein d'un système de calcul formel prend la forme d'une succession d'étapes de calcul formel et de calcul numérique. Sur ce principe, ne doit-on pas désormais envisager TOUTE simulation comme l'alternance d'étapes de calcul formel et d'étapes de calcul numérique ?

Dans ce but, le Chapitre 2 établit un modèle structurel et comportemental pour la coopération entre des systèmes de calcul formel et des systèmes de calcul numérique.

Chapitre 2. Un modèle de coopération entre calcul formel et calcul numérique

La simulation numérique vise principalement deux objectifs :

1. La **fiabilité des résultats** produits ;
2. La **performance de la solution informatique** réalisant la simulation, afin que cette dernière concurrence avantageusement la validation grandeur nature, ou la construction d'un pilote, tant du point de vue du coût que du point de vue du temps de mise en œuvre.

Ces objectifs sont a priori antinomiques. A titre d'exemple, si le calcul en virgule flottante en quadruple précision peut dans certains cas[6] apporter plus de fiabilité à un résultat de simulation, l'arithmétique réelle en quadruple précision requiert le plus souvent des temps d'exécution plus importants qu'une arithmétique réelle en double précision.

Ce chapitre propose d'intégrer des éléments du calcul formel dans la chaîne de simulation numérique afin de contribuer à la fois à l'objectif de fiabilité et à l'objectif de performance. Mais contrairement aux approches détaillées au Chapitre 1, consistant soit à faire du calcul numérique au sein d'un système de calcul formel, soit à utiliser le calcul formel comme un pré-processeur du calcul numérique, dans le travail présenté les éléments de calcul formel interviennent à l'interface entre la spécification et la résolution numérique. Ils prennent place entre les outils de spécification du problème ou de spécification de la solution existants, et les algorithmes de résolution numérique. Si cette contrainte semble réduire l'apport possible du calcul formel en simulation numérique, elle favorise l'adoption des solutions proposées : il ne s'agit pas de remettre en cause des environnements de simulation donnant satisfaction la plupart du temps, mais bien de leur fournir des services, de la façon la plus transparente possible, pour leur apporter un plus en termes de fiabilité ou de performance. La conception du système présenté ici applique *« un modèle de coopération de type complémentaire »*, détaillé dans (Zaraté 2005), caractérisé par le fait qu'aucune interférence n'existe entre les tâches affectées aux différents acteurs du système global[7].

[6] Le chapitre 1.1.2.2 énonce quelques propriétés « intuitives » mais fausses de l'arithmétique en virgule flottante.

[7] (Zaraté 2005) distingue trois types de coopération : complémentaire, interdépendante et négociée.

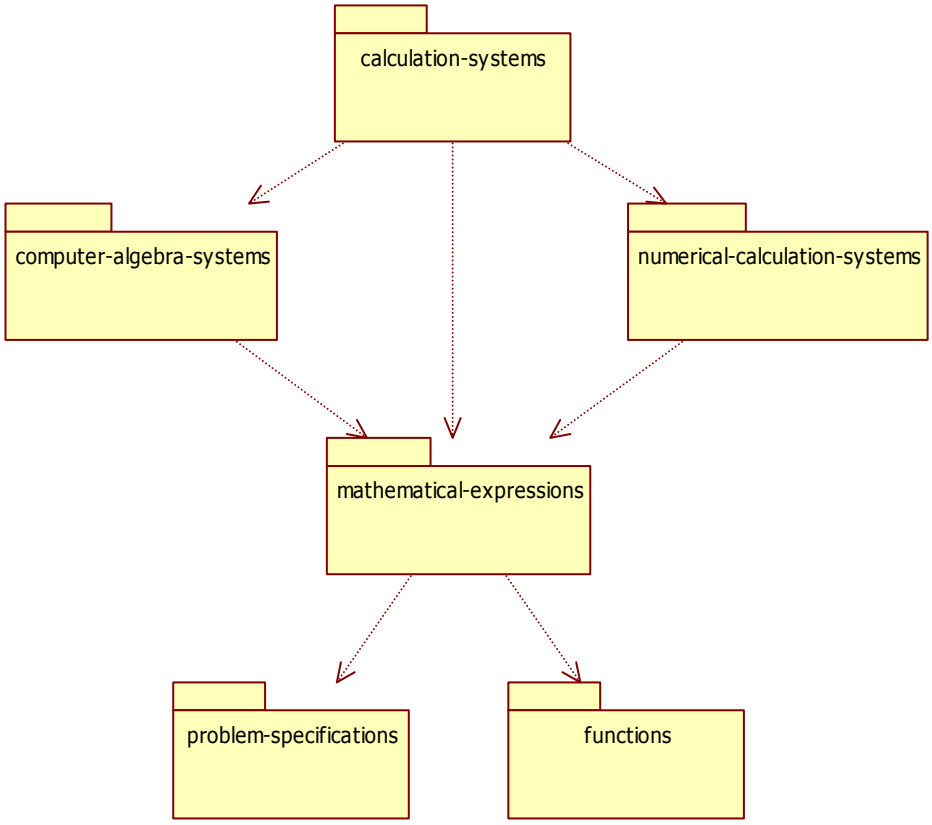

Figure 29 – Paquetages définis lors de la conception du modèle de coopération entre calcul formel et calcul numérique.

La partie 2.1 décrit la structure et le comportement du sous-système de construction et d'évaluation d'expressions mathématiques comportant des nombres et des symboles. Selon le diagramme de la Figure 29, elle correspond à l'étude des interactions entre des entités définies dans les paquetages **computer-algebra-systems** et **numerical-calculation-systems** avec des entités définies dans le paquetage **mathematical-expressions**. A partir de ce sous-système, la partie 2.2 construit un sous-système de résolution symbolico-numérique d'équations continues permettant de calculer une solution réelle unique d'un système d'équations linéaires ou non linéaires, d'un système d'équations algébro-différentielles, ou d'un problème d'optimisation sous contraintes non linéaires. Les entités de ce sous-système sont regroupées au sein d'un paquetage **problem-specifications**. Le système de calcul symbolico-numérique proposé est obtenu par l'ajout, dans la partie 2.3, d'entités modélisant des fonctions implicites définies par un système d'équations continues ou par un problème d'optimisation. Dans le modèle de conception proposé, ces entités apparaissent dans un paquetage **functions**. La partie 2.4 complète ce modèle par des apports complémentaires permettant d'une part la réutilisation de codes de calcul existants, d'autre part la réutilisation performante d'expressions mathématiques, déjà créées, voire déjà évaluées.

2.1. Evaluation symbolico-numérique d'expressions mathématiques

La partie 2.1.1 propose un modèle général d'un système de calcul formel, qui sera exploité ensuite pour construire le modèle du système de calcul symbolico-numérique proposé. La partie 2.1.1.1 établit un modèle structurel de système de calcul formel. Elle isole les principales entités d'un tel système, les met en relation, et détaille les services fournis par chacun des éléments constitutifs. La partie 2.1.1.2 analyse les principaux aspects du modèle comportemental commun, et rappelle la cause première des différences comportementales entre les systèmes de calcul formel : le système de règles de transformations formelles des expressions.

2.1.1. Modèle d'un système de calcul formel existant

2.1.1.1. Modèle structurel

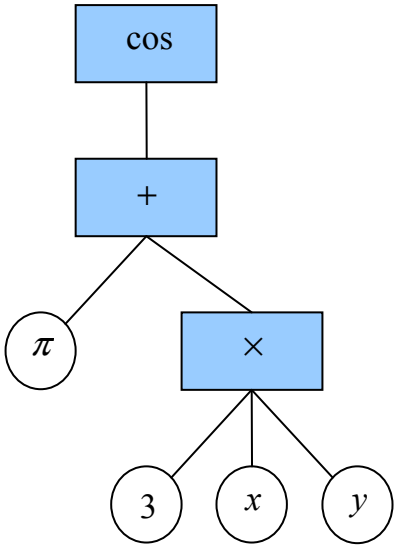

Figure 30 – Une structure de données arborescente pour la représentation d'expressions mathématiques.

La Figure 30 donne l'exemple d'une structure de données arborescente associée à l'expression mathématique $\cos(\pi + 3 \times x \times y)$. Dans de tels arbres, les feuilles sont les expressions atomiques, nombres et symboles, les nœuds intermédiaires sont des opérateurs arithmétiques ou des fonctions mathématiques dont les sous-expressions sont les arguments. Ces arbres peuvent être générés par un compilateur, ou un interpréteur, lors de l'analyse syntaxique d'une expression mathématique écrite dans un langage informatique particulier. Les feuilles sont alors des nombres ou des variables numériques, les nœuds intermédiaires sont des opérateurs arithmétiques, des fonctions mathématiques du langage, ou des fonctions mathématiques définies

par ailleurs. Les arguments de ces opérateurs ou de ces fonctions sont des nombres, ainsi que les valeurs calculées.

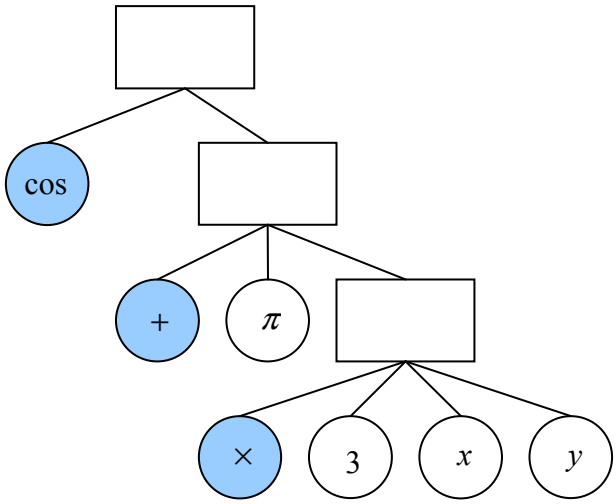

Figure 31 – Une structure de données arborescente pour la représentation d'expressions mathématiques en calcul formel

La représentation arborescente précédente des expressions mathématiques n'est pas adaptée aux systèmes de calcul formel. En imposant aux opérateurs arithmétiques et aux fonctions mathématiques d'être des nœuds intermédiaires de la structure arborescente, elle leur interdit d'être eux-mêmes des feuilles, c'est-à-dire des arguments d'autres opérateurs. La prise en compte d'expressions mathématiques telles que la composition $\cos \circ \log$ de deux fonctions, ou la dérivée partielle $D_i f$ d'une fonction f par rapport à une variable de rang i, impose que les nœuds intermédiaires de la structure arborescente puissent être des expressions quelconques, et que les feuilles de la structure arborescente puissent être des symboles représentant des opérateurs arithmétiques ou des fonctions mathématiques quelconques. A la représentation de l'expression $\cos(\pi + 3 \times x \times y)$ proposée à la Figure 30, un système de calcul formel préfèrera donc la représentation arborescente de la Figure 31, où un opérateur arithmétique ou une fonction mathématique est représenté comme une partie d'une expression, au même titre qu'un argument.

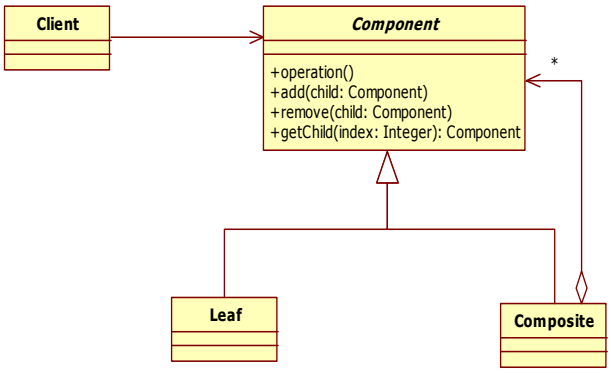

Figure 32 – Modèle de conception « Composite ».

La structure de données arborescente, caractéristique des expressions mathématiques, se conforme tout à fait au modèle de conception « composite », référencé dans (Gamma, Helm, Johnson, & Vlissides 1999). Ce modèle de conception, rappelé dans la Figure 32, *« compose des objets en des structures arborescentes pour représenter des hiérarchies composant/composé. Il permet au client de traiter de la même et unique façon les objets individuels et les combinaisons de ceux-ci »*.

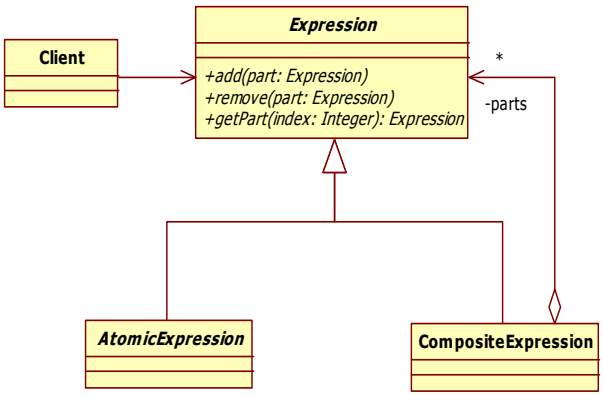

Figure 33 – Modèle de conception « Composite » appliqué aux expressions mathématiques.

La Figure 33 applique le modèle « Composite » au cas particulier des expressions mathématiques. Une expression mathématique est soit composite, soit atomique. Une expression composite comprend des sous-expressions **parts** en nombre quelconque, éventuellement aucune. Les expressions atomiques sont des symboles ou des nombres. Toute expression peut être évaluée, l'évaluation produisant une nouvelle expression. Ce modèle structurel, inhérent à la nature des données à représenter, ne fait pour l'instant aucune hypothèse particulière relative au système présenté. Il est indépendant des outils manipulant les expressions mathématiques : systèmes de calcul formel ou numérique, éditeurs d'équations, etc.

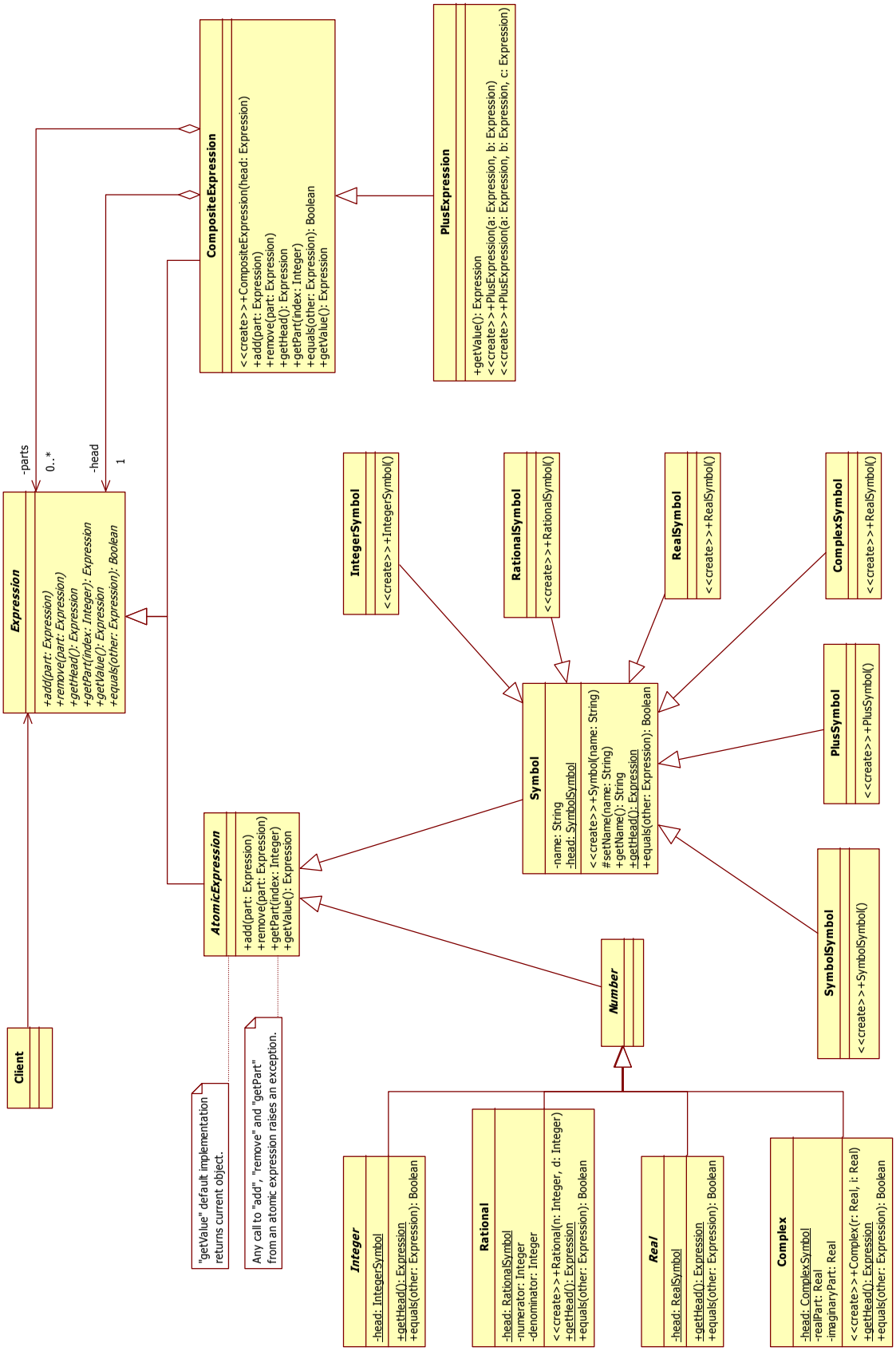

Figure 34 – Modèle structurel des expressions mathématiques pour le calcul formel.

La Figure 34 spécifie un modèle structurel d'expressions mathématiques adapté au calcul formel. Outre le fait que toute expression composite est composée à partir d'un certain nombre, éventuellement nul, d'autres expressions, elle comprend toujours une expression particulière,

head, correspondant à l'opérateur appliqué aux différents parties de l'expression. Contrairement au modèle précédent **head** peut référencer tout type d'expression, en particulier une expression composite. Les opérateurs arithmétiques et les fonctions usuelles pouvant être des arguments dans une expression mathématique formelle, ceux-ci sont définis en tant que classes héritières de la classe **Symbol**. Enfin, un traitement uniforme des expressions consiste à considérer que toute expression, composite ou atomique, est formée par application de l'expression **head** à la liste ordonnée d'arguments **parts**. Cette approche fonctionnelle de bout en bout impose la définition de classes de symboles supplémentaires **IntegerSymbol**, **RealSymbol**, **RationalSymbol**, **ComplexSymbol** et **SymbolSymbol**, qui préfixeront respectivement chaque entier, chaque réel, chaque rationnel, chaque complexe et chaque symbole. Mathematica, Maple et Maxima notamment ayant adopté cette conception[8], nous l'intégrons dans le modèle général des expressions mathématiques formelles présenté.

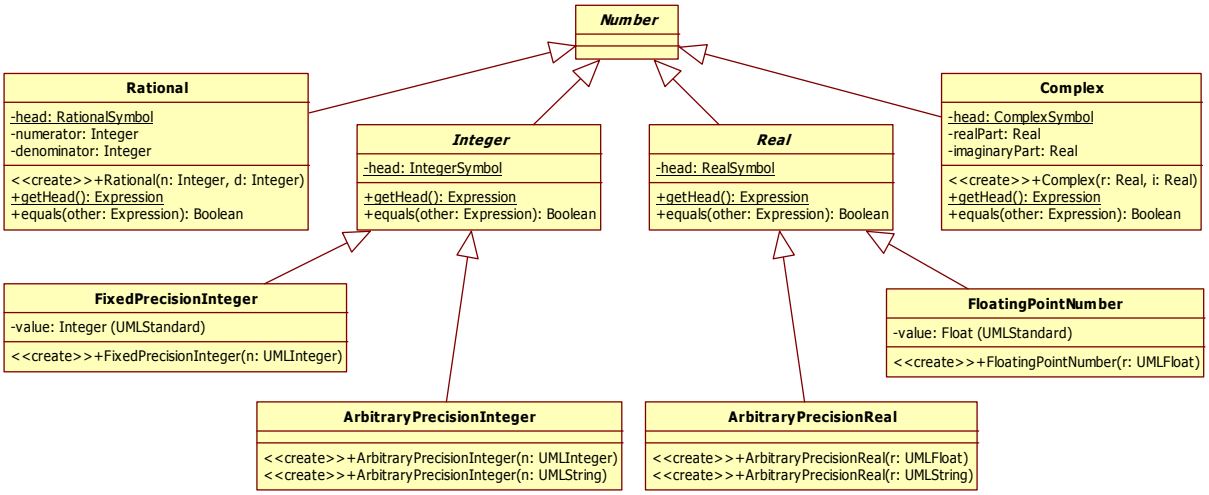

Figure 35 – Modèle structurel des types numériques pour le calcul en précision machine et en précision arbitraire.

Comme le rappelle la synthèse de la partie 1.4, les systèmes de calcul formel disposent d'une arithmétique rationnelle exacte et d'une arithmétique réelle en précision multiple. Par ailleurs, les environnements de calcul numérique peuvent s'appuyer sur des bibliothèques de calcul en précision arbitraire. Le diagramme de classes de la Figure 34 est donc incomplet. La Figure 35 lui ajoute deux classes concrètes, **ArbitraryPrecisionInteger** et **ArbitraryPrecisionReal**, afin de représenter au mieux les expressions mathématiques à partir desquelles il est possible de calculer en précision machine ou en précision arbitraire.

[8] Ces systèmes de calcul formel considèrent **head** comme la partie d'indice 0 d'une expression. Mathematica et Maple définissent la partie d'indice 0 de toute expression, atomique ou composite, via l'ajout de symboles particuliers associés aux types élémentaires. Maxima définit la partie 0 des expressions composites uniquement.

Tous les systèmes de calcul formel manipulent des expressions mathématiques, qu'il est possible de considérer comme une famille d'objets interdépendants, comme cela vient d'être vu. L'ensemble des services communs offerts par les différents systèmes de calcul formel est bien identifié. Il s'agit de :

1. créer les expressions atomiques du système ;
2. combiner les expressions créées pour produire des expressions composites ;
3. évaluer les expressions créées.

De tels systèmes peuvent avantageusement être modélisés à l'aide d'un modèle de conception particulier : la « Fabrique Abstraite ». Ce dernier distingue les classes abstraites, représentant les types d'objets possibles dans le système et les services associés, des classes concrètes à partir desquelles sont effectivement créés les objets de chaque système particulier, et réalisés les services spécifiés dans les classes abstraites.

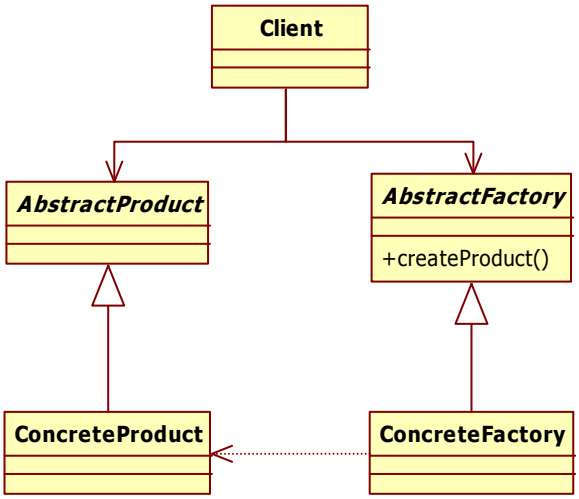

Figure 36 – Modèle de conception « Fabrique Abstraite ».

Plus précisément, en adoptant la nomenclature de la Figure 36, *« AbstractFactory déclare une interface contenant les opérations de création d'objets produits abstraits. ConcreteFactory implémente les opérations de création d'objets produits concrets. AbstractProduct déclare une interface pour un type d'objet produit. ConcreteProduct définit un objet produit qui doit être créé par la fabrique concrète correspondante, et implémente l'interface de AbstractProduct. Client n'utilisera que les interfaces déclarées par les classes AbstractFactory et AbstractProduct. »*

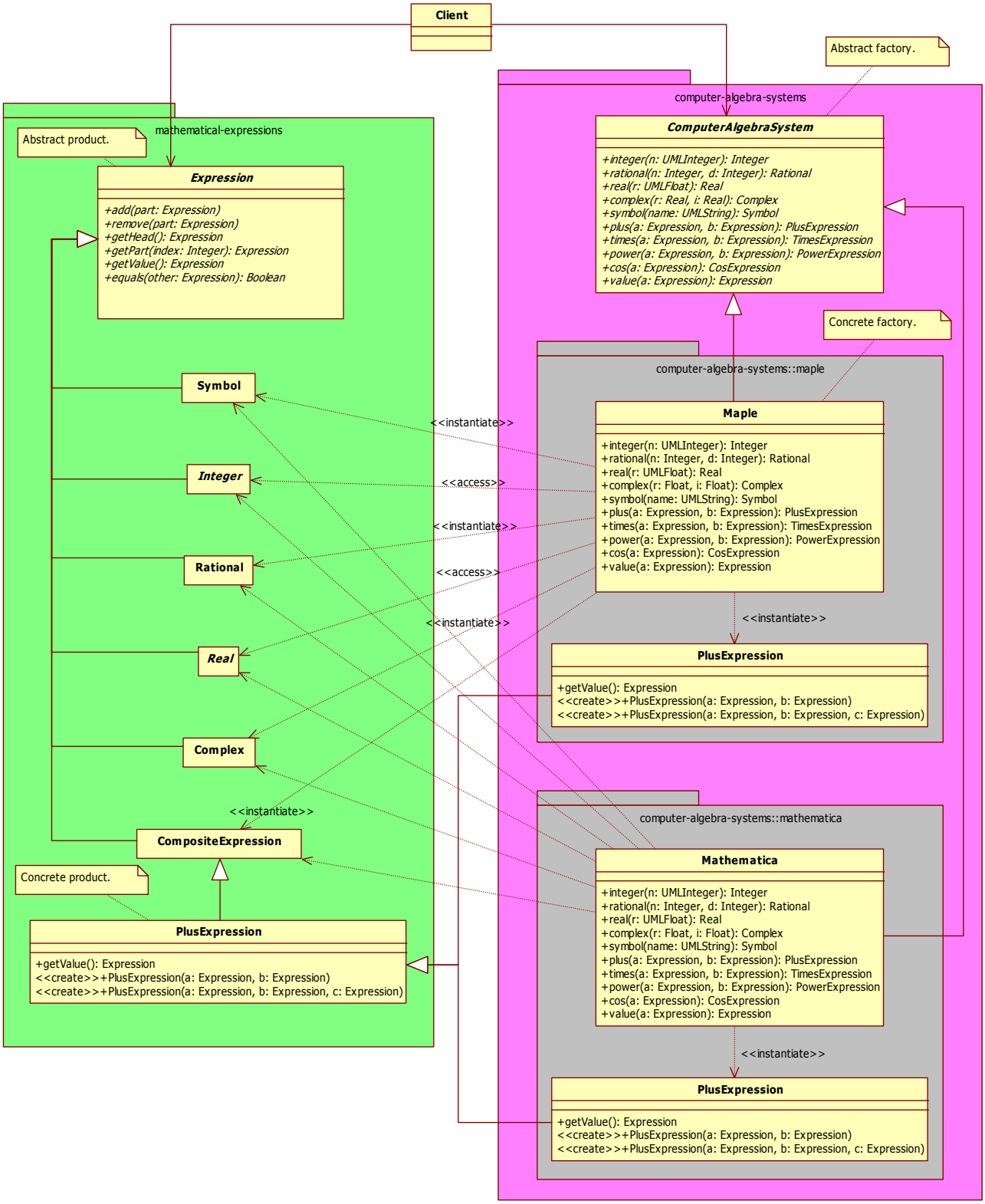

Figure 37 - Modèle de conception « Fabrique Abstraite » appliqué à un système de calcul formel.

Par application du modèle de conception « Fabrique Abstraite », la Figure 37 propose une modélisation structurelle d'un système de calcul formel, où les types et les services communs à tous les systèmes de calcul formel sont isolés dans les classes abstraites **ComputerAlgebraSystem** et **Expression**, tandis que des classes concrètes **Maple**, **Mathematica**, **maple::PlusExpression** et **mathematica::PlusExpression** mettent en œuvre ces types et ces services différemment suivant le système particulier. Comme cela a été vu dans la partie 1.1.1, les stratégies d'évaluation

numérique de Maple et Mathematica sont différentes : l'évaluation par défaut d'une même expression mathématique par Maple et Mathematica peut produire deux expressions mathématiques différentes, non seulement du point de vue de la syntaxe, mais également de la sémantique. Les classes **maple::PlusExpression** et **mathematica::PlusExpression** isolent les spécificités de chacun des systèmes de calcul formel dans son évaluation d'une somme. La méthode **getValue** de la classe **mathematical-expressions ::PlusExpression** propose une mise en œuvre par défaut de l'addition regroupant toutes les règles d'évaluation communes à tous les systèmes : addition de deux symboles identiques, addition de deux entiers opposés, addition de deux entiers en précision arbitraire, etc. Chaque méthode **getValue** d'une classe fille de **mathematical-expressions ::PlusExpression** complète les règles d'évaluation communes par un ensemble de règles d'évaluation spécifiques du système de calcul formel considéré. Il en va de même pour toutes les classes associées à l'évaluation d'un type particulier d'expressions : **TimesExpression**, **PowerExpression**, **CosExpression**, etc.

Le modèle structurel de système de calcul formel apporte la première spécificité du calcul formel. Contrairement à un éditeur d'équations par exemple, les expressions mathématiques y sont non seulement composées mais aussi évaluées, comme dans un système de calcul numérique. Si le résultat de l'évaluation d'une expression mathématique par un système de calcul numérique est toujours un nombre, le résultat de l'évaluation d'une expression mathématique par un système de calcul formel est, plus généralement, une expression mathématique.

Le système de calcul formel constitue l'entité permettant de produire les expressions atomiques, symboles et nombres, ainsi que les expressions composites par appel de fonctions ou d'opérateurs mathématiques. Seuls quelques unes des fonctions accessibles dans tous les systèmes de calcul formel figurent explicitement dans ce modèle. L'interface de la classe **ComputerAlgebraSystem** suggère que toute expression peut être vue comme l'application d'une fonction à des arguments. L'évaluation d'une expression par la méthode **value** consiste alors à calculer le résultat de l'application de la fonction aux arguments. Le lien fort entre le calcul formel et la programmation fonctionnelle, déjà noté dans 1.1.4.4 et dans 1.4, est confirmé.

2.1.1.2. Modèle comportemental

Le modèle comportemental d'un système de calcul formel est d'abord illustré dans la partie 2.1.1.2.1 par la description de deux scénarii d'évaluation d'une même expression mathématique par Maple et par Mathematica. Ceux-ci permettront de répertorier certains points communs et certaines différences entre les systèmes de calcul formel. En adoptant le modèle

structurel décrit dans la partie précédente, la sémantique d'un système de calcul formel est principalement mise en œuvre :

1. dans les méthodes de construction et d'accès aux expressions mathématiques déclarées dans la classe **ComputerAlgebraSystem** : **integer**, **real**, **cos**, **plus**, etc. ;
2. dans la méthode **value** de la classe **ComputerAlgebraSystem**, méthode d'évaluation d'une expression quelconque ;
3. dans la méthode **getValue** de la classe **CompositeExpression** ;
4. dans les méthodes **getValue** déclarées dans les classes **PlusExpression**, **TimesExpression**, etc., associées à l'évaluation des différents types d'expressions.

Les parties 2.1.1.2.2, 2.1.1.2.3, 2.1.1.2.4, 2.1.1.2.5 détaillent respectivement ces méthodes, et exhibent un modèle comportemental par défaut que chaque système de calcul formel vient enrichir ou modifier par redéfinition d'une ou plusieurs de ces méthodes. Le modèle comportemental par défaut est pris comme point de départ du système de calcul symbolico-numérique proposé.

2.1.1.2.1. Scénarii d'évaluation d'une expression mathématique donnée

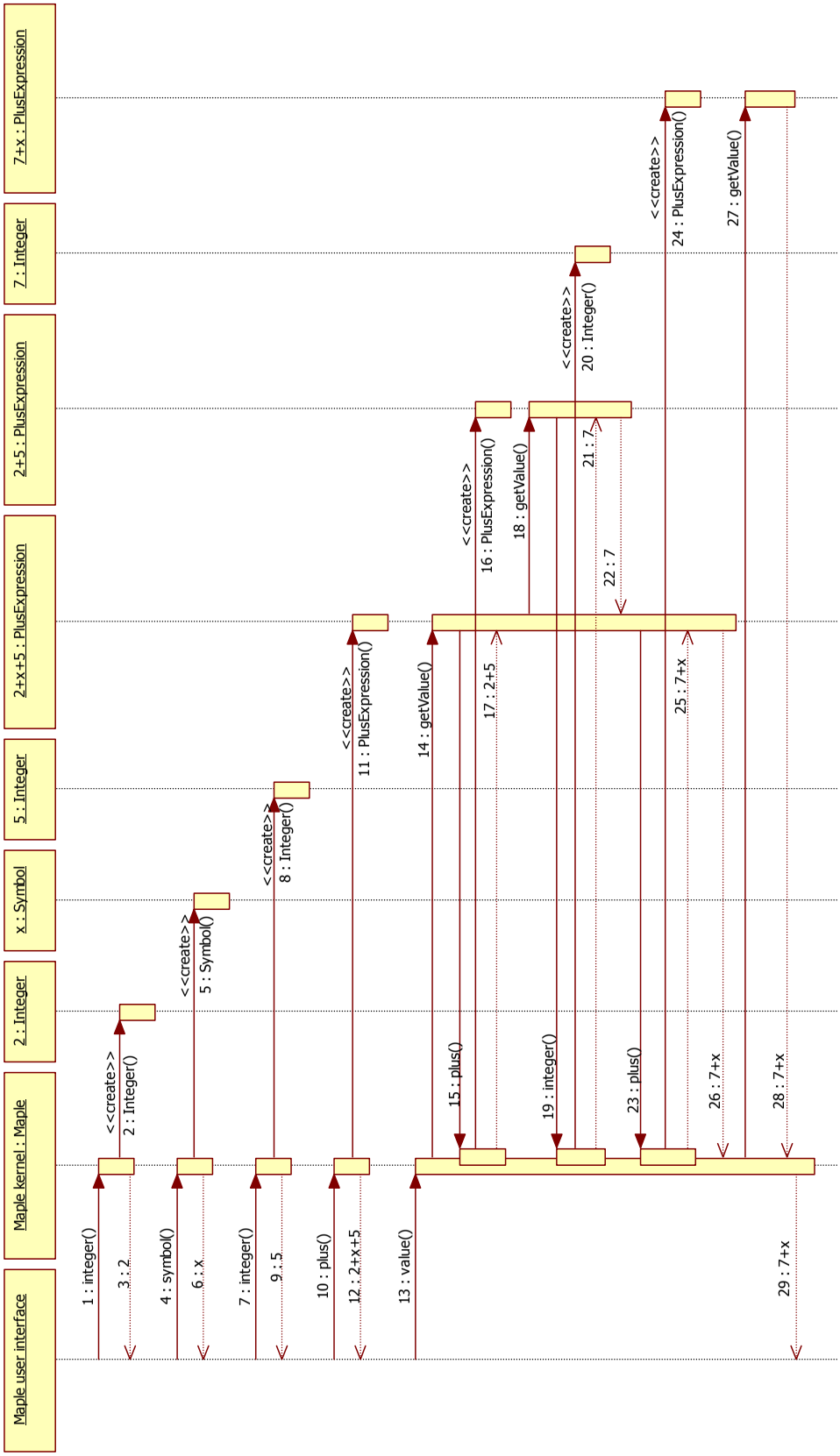

Figure 38 – Scénario d'évaluation de l'unique expression $2+x+5$ par Maple.

Le diagramme de séquence de la Figure 38 représente un scénario d'évaluation de l'expression mathématique $2+x+5$ par le système de calcul formel Maple. Ce scénario fait l'hypothèse que seule cette expression est évaluée lors d'une session. Les appels 1, 4, 7 et 10 déclenchent respectivement la création du nombre entier 2, du symbole x, du nombre entier 5 et enfin de l'expression $2+x+5$. L'appel 13 de la méthode **value** à partir de l'objet « Maple kernel » déclenche l'appel 14 de la méthode **getValue** de l'objet « $2+x+5$ ». Cette méthode retourne en 29 un objet « $7+x$ ». Plus précisément, l'objet « $2+x+5$ » fait appel à la fabrique concrète « Maple kernel » pour créer les objets « $2+5$ », « 7 » et « $7+x$ ». Les appels 15, 19 et 23 demandent au système de calcul formel l'accès à certaines expressions mathématiques. Le système de calcul formel crée les expressions demandées en 16, 20 et 24, et les retourne en 17, 21 et 25. Le point principal à noter est que l'objet « $2+x+5$ » demande l'accès à certaines expressions au système de calcul formel.

Le modèle structurel de la Figure 37 est incomplet. Une expression créée à partir d'un système de calcul formel doit référencer ce dernier. Cette référence se traduit par une association de la classe **Expression** vers la classe **CalculationSystem** dans le modèle structurel.

Le but de ce référencement pour une expression donnée est de pouvoir demander au système de calcul formel qui l'a créée l'accès à d'autres expressions requises par la méthode **getValue**.

Le scénario d'évaluation s'achève avec l'appel en 27 de la méthode **getValue** de l'objet « $7+x$ ». Aucune règle ne permettant de transformer l'expression $7+x$, le résultat 28 de l'appel précédent est l'objet lui-même. Le résultat 29 de l'évaluation de l'expression $2+x+5$ par Maple est donc l'expression $7+x$.

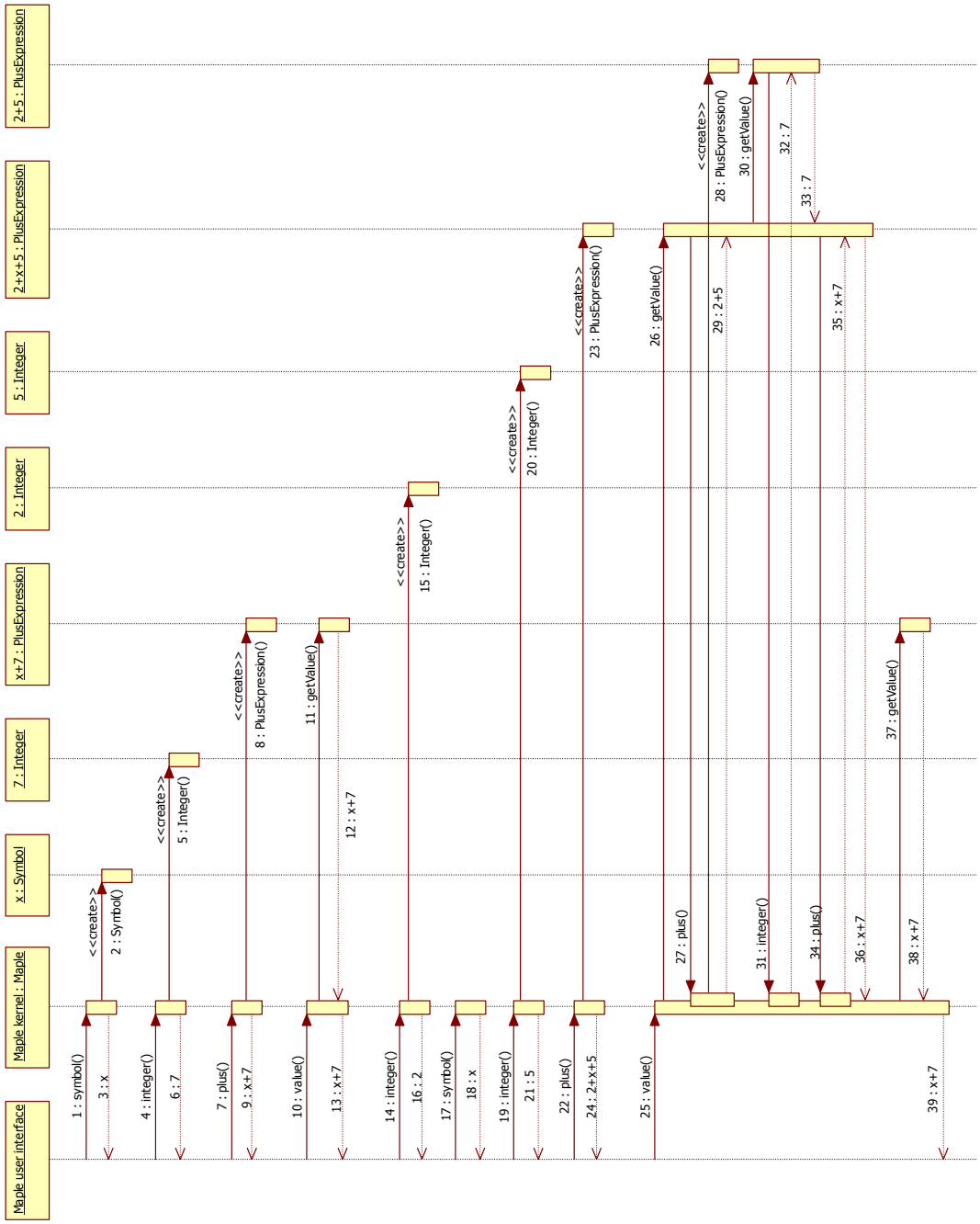

Figure 39 – Scénario d'évaluation des seules expressions $x+7$ et $2+x+5$ par Maple.

Le diagramme de séquence de la Figure 39 représente un autre scénario d'évaluation de l'expression mathématique $2+x+5$ par le système de calcul formel Maple. Dans ce cas, la session se limite à l'évaluation successive des deux expressions $x+7$ et $2+x+5$. Les stimuli UML 1 à 13 consistent à construire et à retourner l'expression $x+7$. Les stimuli 14 à 39 construisent et évaluent l'expression $2+x+5$. L'évaluation par Maple de l'expression $2+x+5$ retourne l'expression $x+7$. Le scénario courant conduit donc à une expression $x+7$ structurellement différente, bien qu'égale, de l'expression $7+x$ produite lors du scénario décrit dans la Figure 38. L'appel 34 de la méthode **plus** de l'objet « Maple kernel » demande l'accès à un objet correspondant à l'expression $7+x$. Le système de calcul formel dispose déjà de l'objet « $x+7$ » qu'il a construit et évalué précédemment. Le stimulus 35 retourne donc ce dernier.

Le modèle structurel de la Figure 37 est incomplet. Un système de calcul formel référence certaines expressions créées auparavant. Cette référence se traduit par le fait que l'association, déjà définie de la classe **Expression** vers la classe **CalculationSystem** dans le modèle structurel, devient bidirectionnelle. La cardinalité du rôle supplémentaire est 0..*, un système de calcul pouvant référencer un nombre quelconque d'expressions.

L'un des buts de ce référencement est la réutilisation ultérieure des expressions disponibles pour la construction de nouvelles expressions. Ce point fera l'objet de la partie 2.1.1.2.2.

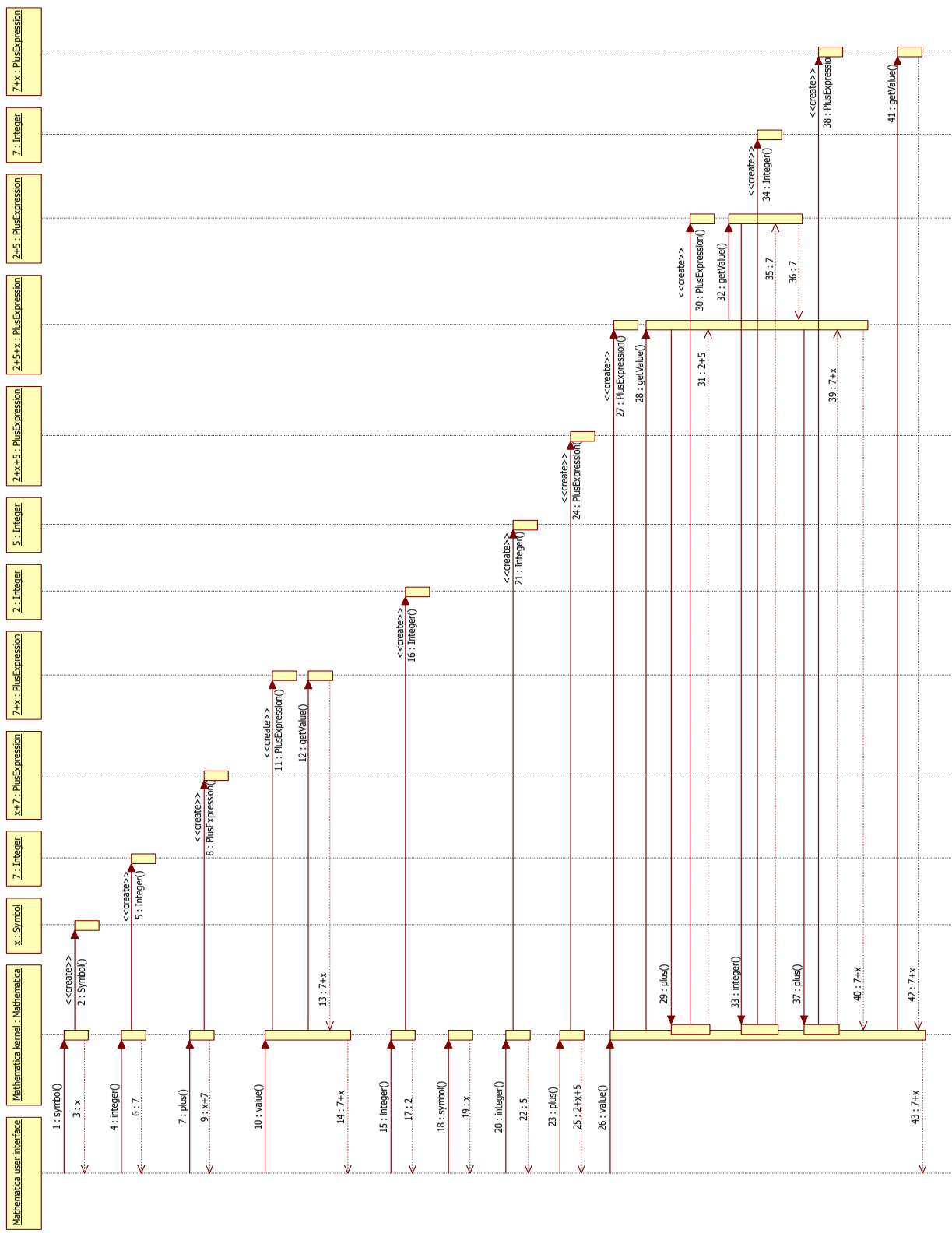

Figure 40 – Scénario d'évaluation des seules expressions $x+7$ puis $2+x+5$ par Mathematica.

Le scénario d'évaluation des seules expressions $x+7$ puis $2+x+5$ joué dans le système de calcul formel Mathematica fait l'objet de la Figure 40. Alors que les évaluations successives par Maple de ces deux expressions produisaient toutes deux le résultat $x+7$, Mathematica évalue ces expressions en $7+x$. La raison de ces résultats tient à l'adoption par Mathematica d'une forme canonique associée aux expressions $x+7$ et $7+x$. L'opérateur + possède l'attribut **`Orderless`**, qui signifie sa propriété de commutativité. Par conséquent, lors d'une

évaluation par la méthode **value** d'une expression dont $+$ est la tête[9], les arguments[10] de cet opérateur sont systématiquement ordonnés selon une relation d'ordre prédéfini. Dans le cas du diagramme de séquence de la Figure 40, les stimuli 11 et 27 réordonnent les arguments des expressions $x+7$ et $2+x+5$ respectivement. Dès lors, l'évaluation par Mathematica des expressions $x+7$ et $7+x$ retourne toujours l'expression $7+x$.

Deux scénarii d'évaluation par Maple et Mathematica d'expressions élémentaires ont suffi à montrer un point commun à tous les systèmes de calcul formel -l'interaction forte entre un système de calcul formel et les expressions mathématiques créées par ce système- et des différences de stratégie entre ceux-ci, notamment dans la démarche de réutilisation d'expressions déjà créées ou dans l'adoption de formes canoniques. Les parties suivantes précisent ces points.

2.1.1.2.2. Construction et accès aux expressions mathématiques

A l'exception de la méthode **value**, les méthodes déclarées dans la classe abstraite **ComputerAlgebraSystem** ont pour rôle l'accès aux expressions mathématiques. Ainsi en désignant par **cas** une instance d'un système de calcul formel particulier, **cas.integer(1)** retourne-t-elle l'entier un considéré comme une expression, **cas.symbol('x')** le symbole de nom « x » considéré comme une expression, et **cas.plus(cas.integer(1), cas.symbol('x'))** l'expression $1+x$. Excepté s'il s'agit d'un symbole prédéfini du système, un appel à ces méthodes conduit à la création d'une expression particulière retournée comme résultat de l'appel. Selon le modèle de conception « fabrique abstraite », l'appel d'une méthode de la fabrique concrète déclenche l'appel d'un constructeur de produit concret. Dans le scénario de la Figure 38, cette situation correspond par exemple à l'appel 1 de la méthode **integer** de l'objet « Maple kernel » qui déclenche la création de l'objet « 2 ».

Toutefois, un appel à une méthode d'accès à une expression ne conduit pas toujours à la création d'une nouvelle expression. Tous les systèmes de calcul formel créent une instance unique d'un symbole de nom donné. En effet, ces symboles, outre leur nom, référencent toujours une information riche : les définitions associées au symbole dans Mathematica et Maple, les attributs éventuels du symbole dans Mathematica, des valeurs déjà calculées de la fonction désignée par le symbole dans Maple, etc. Plutôt que de maintenir plusieurs instances d'un même

[9] Attribut **head** de la classe **CompositeExpression** selon le modèle structurel de la Figure 34.

[10] Attributs **parts** de la classe **CompositeExpression** selon le modèle structurel de la Figure 34.

symbole référençant les mêmes informations, il est beaucoup plus judicieux de maintenir un exemplaire unique de chacun des symboles.

Certains systèmes de calcul formel maintiennent, outre la liste des symboles déjà créés, une liste d'autres expressions construites auparavant, qu'ils réutilisent lors de la construction de nouvelles expressions. Ce partage de sous-expressions est très voisin du processus d'élimination des sous-expressions communes vu en 1.2.4.3. Le but est de limiter la taille en mémoire des expressions manipulées par le système et d'éviter éventuellement, comme lors de l'élimination des sous-expressions communes, le calcul répété d'une même expression. L'objectif de non-réévaluation d'une même expression impose, outre le partage des sous-expressions communes, un mécanisme de cache d'évaluation qui maintienne le résultat de la dernière évaluation et permette de juger de sa validité lors d'une réutilisation éventuelle. Les stratégies de partage des sous-expressions sont différentes d'un système de calcul formel à un autre.

Comme le précise (Giusti et al. 2000) *« le système de calcul formel Maple est basé sur un partage systématique des sous-expressions communes. Des objets qui pourraient apparaître comme des arbres d'expressions à l'utilisateur sont en fait stockés comme des graphes orientés acycliques, où seulement une copie de chaque sous-arbre distinct est conservée. Ceci est réalisé en maintenant une table de hachage de toutes les expressions présentes simultanément dans une session. La structure ainsi obtenue peut être vue comme un unique graphe orienté acyclique dont les enfants de la racine correspondent à toutes les sous-expressions résidant simultanément en mémoire. »*

A l'inverse de Maple, par défaut Mathematica ne partage pas les sous-expressions communes à l'exception des symboles. Le partage des sous-expressions communes intervient uniquement sur des expressions déjà construites, et à la demande de l'utilisateur. La commande **Share** partage toutes les sous-expressions communes présentes dans une expression unique ou bien dans l'ensemble des expressions de la session .

2.1.1.2.3. Evaluation par défaut des expressions composites

La méthode **getValue() :Expression**, déclarée dans la classe abstraite **Expression** possède une sémantique par défaut, définie dans la classe **CompositeExpression**. Evaluer une expression composite **e** à l'aide de l'appel **e.getValue()** consiste à évaluer la tête de l'expression à l'aide de l'instruction **e.getHead().value()**, puis à évaluer chacune des parties de **e** à l'aide d'instructions de la forme **e.getPart(i).value()**. L'expression composite résultat **r** de l'appel **e.getValue()** a le même nombre **e.getLength()** de sous-expressions que l'expression initiale **e**. La tête de l'expression **r** est le résultat de l'appel **e.getHead().value()**, la sous-expression d'indice **i** de l'expression **r** est le

résultat de l'appel **e.getPart(i).value()**. Le code, dans la syntaxe du langage Java[11], de la méthode **getValue** de la classe **CompositeExpression** est donné dans la Figure 41.

```
public Expression getValue() {
    Expression result = new CompositeExpression(getHead().getValue());
    for (int i=1; i<=getLength(); i++) {
        result.add(getPart(i).getValue());
    };
    return result;
}
```

Figure 41 – Méthode **getValue** de la classe **CompositeExpression**.

La sémantique de la méthode **getValue** de la classe **CompositeExpression** est identique d'un système de calcul formel à un autre.

2.1.1.2.4. Règles de transformations formelles associées à chaque type d'expression

Selon que l'expression mathématique à évaluer est une somme ou un produit de termes, la sémantique de la méthode **getValue**, déclarée dans la classe abstraite **Expression**, doit être différente. Les classes héritières de la classe **CompositeExpression** redéfinissent toutes la méthode **getValue**. Plus exactement, chaque fois que des instances d'expressions mathématiques composites ont une sémantique d'évaluation spécifique, une classe héritière de la classe **CompositeExpression** est créée et la méthode **getValue** y est redéfinie. En particulier, à chaque opérateur arithmétique ou à chaque fonction mathématique usuelle est donc associée une sous-classe de la classe **CompositeExpression**. L'opérateur + donne lieu à la création de la classe **PlusExpression**, dont la méthode **getValue** mettra en œuvre l'algorithme d'addition de toutes les expressions mathématiques. La fonction mathématique cos donne lieu à la création de la classe **CosExpression**, dont la méthode **getValue** mettra en œuvre l'algorithme du calcul du cosinus de toute expression mathématique. La nécessité d'une sémantique propre à l'évaluation de groupes d'expressions dont la tête n'est pas un symbole -opérateur ou fonction mathématique- peut également conduire à la création de nouvelles sous-classes de la classe **CompositeExpression**. Cela sera vu ultérieurement.

Les différences comportementales entre les systèmes de calcul formel résultent d'abord des différences entre les algorithmes codés dans les différentes méthodes **getValue** des sous-classes de **CompositeExpression**. Le nombre de règles de transformations formelles codées, leur

[11] On convient, dans tout le manuscrit, d'exprimer le code des méthodes présentées dans la langage Java.

nature et l'ordre d'application de ces règles donne une sémantique particulière à chaque méthode **getValue**, et donc plus globalement au système de calcul formel considéré.

2.1.1.2.5. Boucle d'évaluation

La méthode **value** de la classe **ComputerAlgebraSystem** se distingue des autres méthodes déclarées dans cette classe. Si toutes les autres méthodes retournent toujours un objet dont le type est un sous-type de la classe **Expression**, la méthode value fournit un objet du type **Expression** lui-même. Alors qu'un appel du type **mathematica.plus(x,y)** donne évidemment accès à une expression plus, typée **PlusExpression** selon le modèle structurel de la Figure 37, l'appel **mathematica.value(argument)** produit une expression dont le type exact n'est pas connu statiquement car il dépend non seulement du type dynamique de l'argument, mais encore de sa valeur précise. A titre d'exemple, si **mathematica.value(mathematica.plus(x, y))** retourne l'objet « $x+y$ » de type **PlusExpression**, **mathematica.value(mathematica. plus(x, x))** retourne un objet « $2\times x$ » de type **TimesExpression**.

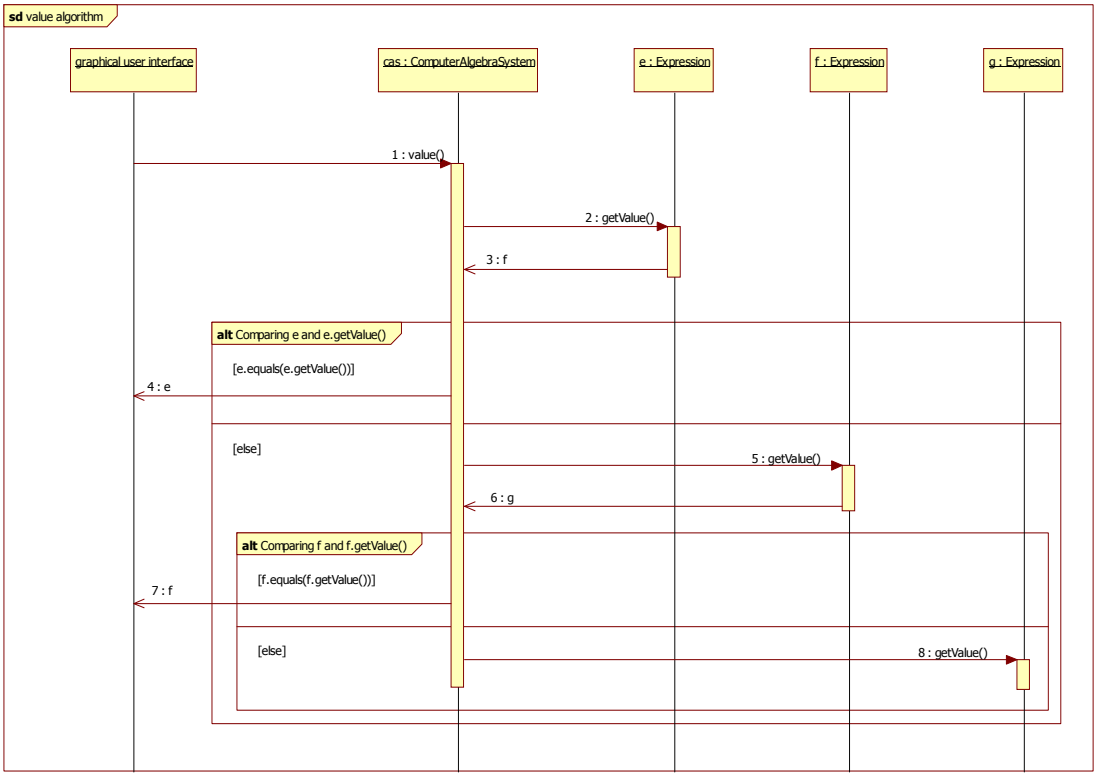

Figure 42 – Algorithme de la méthode **value** de la classe **ComputerAlgebraSystem**.

Une sémantique minimale de la méthode **value** de la classe **ComputerAlgebraSystem**, commune à tous les systèmes de calcul formel peut être exhibée. Etant donné un argument d'appel **e**, **value(e)** procède à l'appel **e.getValue()**. Si le résultat de cet appel est une expression syntaxiquement identique à **e**, i.e. si **e.getValue().equals(e)**, alors le résultat est **e**. Sinon l'expression **e.getValue()** est désigné par **f** et **value** procède à l'appel **f.getValue()**. Si le résultat de cet appel est

une expression syntaxiquement identique à **f**, i.e. si **f.getValue().equals(f)**, alors le résultat est **f**. Sinon le processus itératif est poursuivi selon le diagramme de séquence de la Figure 42 jusqu'à l'obtention d'une expression dont l'évaluation est elle-même. L'exécution de la méthode **value(e)** peut donc être vue comme le calcul d'un point fixe éventuel de la fonction **getValue** avec **e** pour point initial.

La simplicité et la généralité de ce processus générique d'évaluation fait la force des systèmes de calcul formel qui sont d'abord des machines à transformer des expressions mathématiques. Ce processus peut cependant être coûteux en temps d'exécution car il est itératif. Pire encore, il ne garantit pas qu'une évaluation se termine !

2.1.2. Modèle du système de calcul symbolico-numérique proposé

2.1.2.1. Modèle structurel d'un système de calcul

La modélisation structurelle d'un système de calcul formel nous a conduit en 2.1.1.1 à établir une hiérarchie de classes modélisant les expressions mathématiques. Il paraît évident que cette même hiérarchie reste valable lorsque l'on s'intéresse, non plus à des systèmes de calcul formel, mais à des systèmes de calcul numérique tels que les environnements de calcul intégré MATLAB et Scilab, ou les bibliothèques mathématiques IMSL ou Nag. Certes la sémantique associée à l'évaluation de certaines classes d'expressions diffèrera entre systèmes de calcul formel et systèmes de calcul numérique, mais cette sémantique diffère déjà d'un système de calcul formel à un autre !

Les outils de calcul numérique manipulent eux aussi des nombres, parfois en précision multiple, des expressions composites et également des symboles. Les symboles des systèmes de calcul numérique peuvent servir à désigner d'autres expressions mathématiques numériques. Ainsi MATLAB ne manipule-t-il que des nombres ou des matrices de nombres[12], les opérations d'affectation d'une quantité numérique à un symbole servant à nommer la quantité numérique. Les symboles ainsi construits peuvent apparaître dans de nouvelles expressions, mais qui désigneront toujours des quantités numériques. Plus subtile est la notion de symbole dans des langages de simulation orientés équations tels que Modelica. Ceux-ci servent à établir les équations caractéristiques du comportement du système à simuler. Excepté s'il s'agit de paramètres, dont la valeur explicite est donnée, ces symboles ne sont associés à aucune valeur

[12] MATLAB peut cependant être complété par une boîte à outils de calcul symbolique incorporant le noyau de calcul formel Maple.

numérique dans le texte Modelica. Ils semblent donc très proches des symboles manipulés par les systèmes de calcul formel. En fait, chacun de ces symboles sera traduit en une variable numérique d'un code de calcul écrit en langage C++ lors d'un processus de compilation décrit dans (Bunus & Fritzson 2004). Durant ce processus le compilateur applique aux expressions mathématiques diverses transformations formelles, telles qu'une élimination des sous-expressions communes ou des simplifications algébriques. Cependant, lors de la simulation numérique, c'est-à dire lors de l'exécution du code C++ synthétisé puis compilé, seules sont évaluées des variables numériques au sein de fonctions représentant notamment les résidus liés aux équations. La sémantique associée à l'évaluation des expressions comprenant des symboles demeure donc plus riche dans les systèmes de calcul formel puisque, contrairement aux systèmes de calcul numérique, les symboles n'ont pas à désigner de quantités numériques pour être évalués au sein d'une expression mathématique. Néanmoins le modèle structurel des expressions mathématiques pour le calcul formel demeure valide pour les expressions mathématiques du calcul numérique. L'utilisation d'un modèle structurel général des expressions mathématiques semble d'autant plus pertinent que les environnements de modélisation et de simulation numérique actuels permettent une formulation déclarative des équations dont la syntaxe s'apparente à celles proposées par les systèmes de calcul formel.

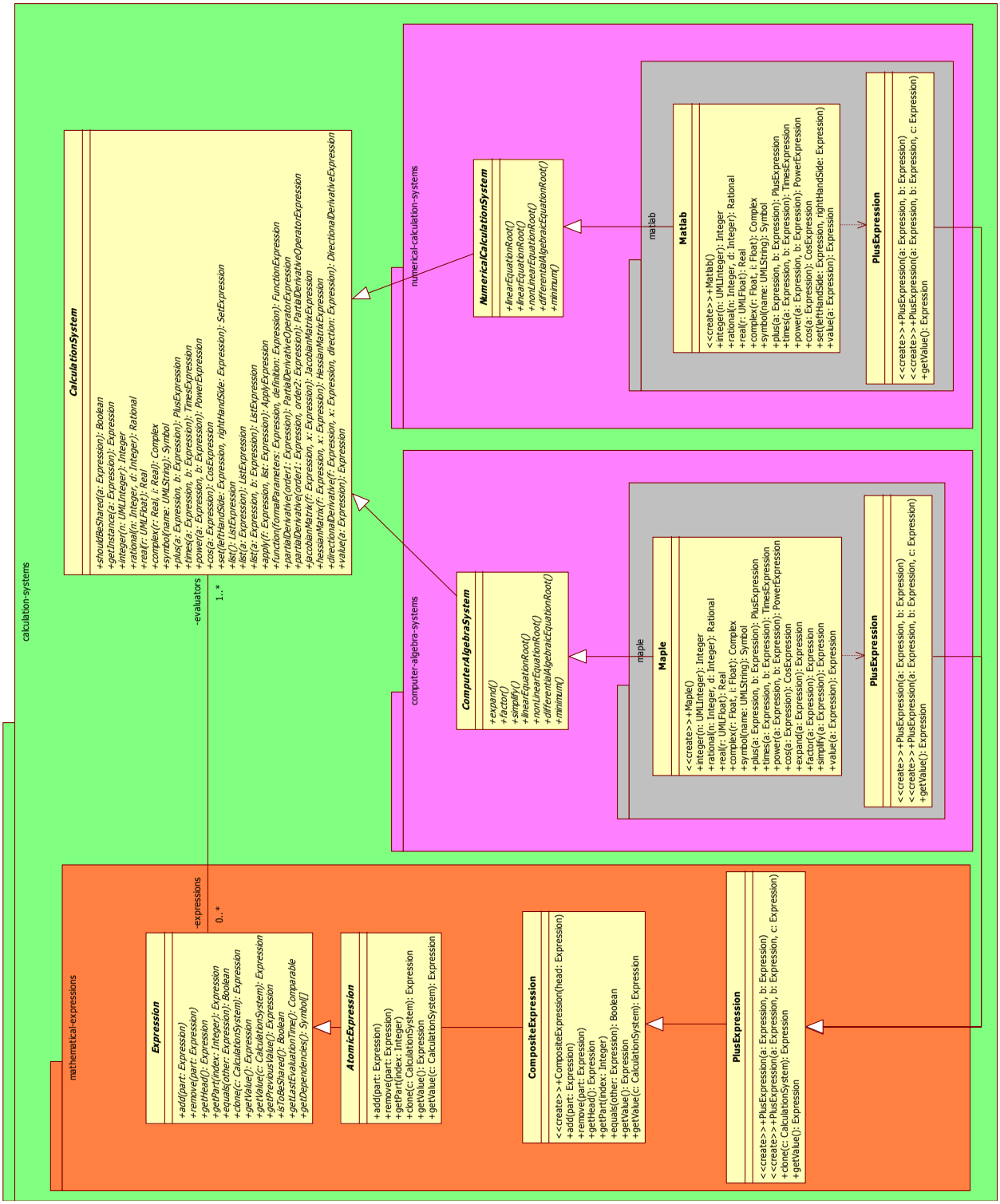

Figure 43 – Modèle structurel d'un système de calcul, formel ou numérique.

Tout comme cela a été vu pour les systèmes de calcul formel dans la partie 2.1.1.1, les systèmes de calcul numérique présentent une interface de fonctions permettant l'accès aux expressions mathématiques manipulées. Une majorité de ces fonctions est commune à tous les systèmes de calcul formel et à tous les systèmes de calcul numérique. Il s'agit des opérateurs arithmétiques et des fonctions mathématiques usuelles. La factorisation de ces méthodes dans la classe **CalculationSystem** conduit au diagramme de classes de la Figure 43. Afin de ne pas surcharger le diagramme de classes, seules quelques méthodes représentatives ont été retenues.

Tout système de calcul comporte, outre les méthodes **plus**, **times** et **power**, des méthodes **subtract** et **divide**. De la même façon la panoplie des fonctions trigonométriques d'un système de calcul ne se réduit évidemment pas à la fonction cosinus mais comprend, la plupart du temps, les fonctions **cos**, **sin**, **tan**, **arccos**, **arcsin**, **arctan**, **cosh**, **sinh**, **tanh**, **arccosh**, **arcsinh**, **arctanh**. L'affectation d'une expression à une autre, modélisée par la méthode **set** de la classe **CalculationSystem** fait également partie de cette interface puisque l'opérateur d'affectation apparaît aussi bien dans les langages utilisés en simulation numérique que dans ceux proposés par les divers systèmes de calcul formel.

Certaines fonctions, dédiées à la transformation d'expressions symboliques, sans considération d'une éventuelle évaluation numérique ultérieure, sont spécifiques des systèmes de calcul formel. A titre d'exemple, la factorisation de polynômes ou le développement d'expressions, respectivement modélisés par les méthodes **factor** et **expand**, appartiennent à cette classe de méthodes. La fonction de simplification des expressions -**simplify**- est également une spécificité des systèmes de calcul formel, qui y appliquent de multiples stratégies afin de réduire la complexité d'une expression selon une métrique choisie.

2.1.2.2. Modèle comportemental d'un système de calcul symbolico-numérique

Au delà des méthodes déclarées dans la classe **ComputerAlgebraSystem**, absentes de la classe **NumericalCalculationSystem**, une différence fondamentale distingue les systèmes de calcul formel des systèmes de calcul numérique : la définition par défaut choisie pour la méthode **value**, déclarée dans la classe **CalculationSystem**. Alors que les systèmes de calcul formel procèdent à des transformations de l'expression fournie en argument jusqu'à obtenir une expression qui ne puisse plus être transformée par une quelconque règle, les systèmes de calcul numérique se contentent d'appliquer une et une seule fois l'algorithme associé au type d'expression à évaluer. L'algorithme par défaut de la méthode **value** de la classe **NumericalCalculationSystem** est simplissime. La Figure 44 le présente.

```
public Expression value(Expression e) {
      return e.getValue();
  }
```

Figure 44 – Méthode **value** de la classe **NumericalCalculationSystem.**

Les considérations précédentes nous ont amené à proposer un modèle structurel général de système de calcul dans lequel s'inscrivent à la fois les systèmes de calcul formel et les systèmes de calcul numérique existants. Cette factorisation maximale des fonctionnalités communes aux différents systèmes de calcul ouvre naturellement la voie au système de calcul

symbolico-numérique proposé. Il s'agit désormais de disposer d'expressions mathématiques et de les évaluer dans le système de calcul de son choix ; soit dans un système de calcul formel, soit dans un système de calcul numérique. L'arbitrage a priori[13] entre les deux critères choisis, la fiabilité et la performance, guidera le choix du système de calcul pour l'évaluation de telle ou telle classe d'expressions.

Dans les systèmes de calcul actuels, qu'ils soient formels ou numériques, la sémantique d'évaluation d'une expression, localisée dans la méthode **getValue** de la classe de l'expression, est toujours liée au système de calcul créateur de cette expression. Ainsi, dans le diagramme de séquence de la Figure 38, détaillant le scénario d'évaluation de l'expression $2+x+5$ par Maple, la méthode **getValue** de la classe **PlusExpression** fait-elle appel au noyau Maple qui a créé l'objet courant pour accéder à l'expression « $2+5$ » dont la sémantique d'évaluation sera également celle de Maple. Rompre cet enfermement sémantique dans un système de calcul unique afin d'exploiter au mieux les qualités de plusieurs systèmes, impose tout d'abord la possibilité de créer à partir d'une expression une autre expression syntaxiquement identique, mais dont la sémantique d'évaluation associée soit celle d'un autre système de calcul. Tel est le but de la méthode **clone**, que nous proposons de déclarer dans la classe **Expression**, et de définir ensuite dans différentes sous-classes. Naturellement, les définitions des méthodes **clone** dans les sous-classes de **Expression** sont constructives. A titre d'exemple la Figure 45 donne le code de la méthode **clone** de la classe **PlusExpression**, en se limitant à la somme de deux opérandes.

```
    public Expression clone(CalculationSystem c) {
        return c.plus(getPart(1).clone(c), getPart(2).clone(c));
    }
```

Figure 45 – Méthode **clone** de la classe **calculation-systems::PlusExpression.**

```
   public Expression getValue() {
        return this;
    }
   public Expression getValue(CalculationSystem c) {
        return this;
    }
```

Figure 46 –Méthodes **getValue** de la classe **AtomicExpression**.

[13] Cet arbitrage a priori définit un modèle de coopération de type complémentaire, mais non évolutif, entre les différents systèmes de calcul.

```
    public Expression getValue() {
        Expression result = new CompositeExpression(getHead().getValue());
        for (int i=1; i<=getLength(); i++) {
            result.add(getPart(i).getValue());
        };
        return result;
    }
   public Expression getValue(CalculationSystem c) {
        return clone(c).getValue());
    }
```

Figure 47 – Méthodes **getValue** de la classe **CompositeExpression**.

La construction d'un clone d'une expression **e** créée à partir d'un système de calcul **c1**, dans un autre système de calcul **c2**, permet dès lors d'évaluer **e** avec la sémantique de **c2**. L'évaluation de l'expression **e**, avec la sémantique du système de calcul **c2**, prend la forme d'un appel à une des deux versions de la méthode **getValue** : **e.getValue(c2)**. Le code des versions par défaut de la méthode **getValue** de la classe **AtomicExpression** est l'objet de la Figure 46. Le code des versions par défaut de la méthode **getValue** de la classe **CompositeExpression** est l'objet de la Figure 47.

La sémantique de la méthode **getValue** ayant été étendue, la sémantique par défaut de la méthode **value** déclarée dans la classe **CalculationSystem** gagne à être étendue également. L'évaluation d'une expression mathématique par un système de calcul consiste désormais à comparer ce dernier et le système de calcul ayant créé l'expression. Si ceux-ci sont identiques, la sémantique par défaut, présentée dans la Figure 42 et la Figure 44 s'applique. Dans le cas contraire, cette sémantique s'applique à un clone de l'expression courante dans le système de calcul courant. La Figure 48 précise l'algorithme par défaut de la méthode **value** définie dans la classe **NumericalCalculationSystem**. Le code proposé suppose la définition d'une méthode **getCreator** dans la classe **Expression**, dont le rôle est de retourner le système de calcul ayant créé l'expression courante. L'algorithme par défaut de la méthode **value**, définie dans la classe **ComputerAlgebraSystem**, est donné dans la Figure 49. Si l'expression est évaluée par un système de calcul qui ne l'a pas créée, alors une copie de cette expression est préalablement créée dans le système de calcul procédant à l'évaluation. Les parties de l'expression sont d'abord évaluées par la méthode **partValues** détaillé dans la Figure 50. Celle-ci produit un objet typé en fonction de l'opérateur arithmétique, ou de la fonction intrinsèque, constituant la tête de l'expression. Cette expression est évaluée selon sa sémantique propre, à l'aide de la méthode **getValue**. Cette évaluation consiste à effectuer une opération sur des opérandes, ou à appliquer une fonction à des arguments. Dans le cas où le résultat de la méthode **getValue** diffère de son argument, le processus d'évaluation complet s'applique de nouveau à la dernière expression produite. Il s'agit

donc d'un processus récursif puisque l'évaluation d'une expression comprend l'évaluation des sous-expressions qui la constituent. Il s'agit également d'un processus itératif puisque des expressions sont évaluées successivement jusqu'à atteindre un point fixe.

Les définitions choisies des méthodes **value** des classes **NumericalCalculationSystem** et **ComputerAlgebraSystem** établissent une coopération entre des systèmes de calcul, formels ou numériques, autour d'un modèle de données commun : les expressions mathématiques. Des stratégies de calcul exploitant les sémantiques d'évaluation distinctes de ces systèmes peuvent dès lors être étudiées.

```
public Expression value(Expression e) {
      if (e.getCreator().equals(this)) {
            return e.getValue();
      } else {
            return e.clone(this).getValue();
      }
    }
```

Figure 48 – Méthode **value** de la classe **NumericalCalculationSystem** pour l'évaluation d'une expression créée par un système de calcul quelconque.

```
    public Expression value(Expression e) {
        Expression partialEvaluationResult, fullEvaluationResult;
        if (e.getCreator().equals(this)) {
            partialEvaluationResult = partValues(e);
        } else {
            partialEvaluationResult = partValues(e.clone(this));
        }
        fullEvaluationResult = partialEvaluationResult.getValue();
        if (fullEvaluationResult.equals(partialEvaluationResult)) {
            return fullEvaluationResult;
        } else {
            return value(fullEvaluationResult);
        }
```

Figure 49 – Méthode **value** de la classe **ComputerAlgebraSystem** pour l'évaluation d'une expression créée par un système de calcul quelconque.

```
    private Expression partValues(Expression e) {
        Expression headValue, result;
        String headValueType;
        int partIndex;
        headValue=value(e.getHead());
        headValueType=headValue.getClass().getName();
        if (headValueType == "PlusSymbol") {
            result = new PlusExpression();
        } else {
            if (headValueType == "TimesSymbol") {
                result = new TimesExpression();
            } else {
                result = new CompositeExpression(headValue);
            }
        };
        for (partIndex=1;partIndex<=e.getLength()[14];partIndex=partIndex+1) {
            result.add(value(e.getPart(partIndex)));
        };
        return result;
    }
```

Figure 50 – Méthode **partValues** de la classe **ComputerAlgebraSystem** pour l'évaluation des parties d'une expression créée par un système de calcul quelconque.

[14] La méthode *getLength*, utilisée lors de l'écriture du code de la méthode *partValues*, ne fait pas partie de la spécification de la classe *Expression*. Cependant toute implémentation associée à la classe *Expression* proposera au moins un mécanisme de parcours de la collection des sous-expressions constituant toute expression.

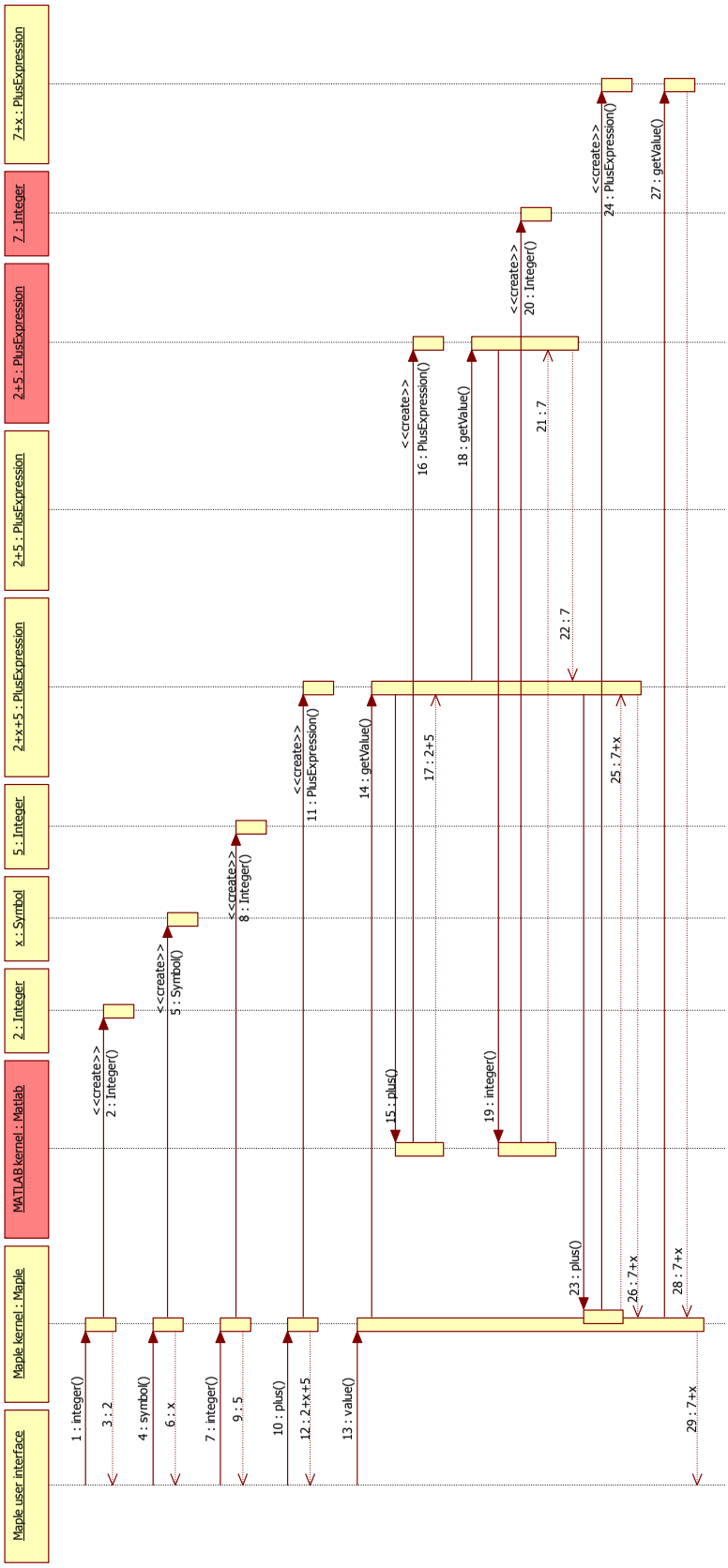

Figure 51 – Scénario d'évaluation conjointe de la seule expression $2+x+5$ par Maple et MATLAB.

Le diagramme de séquence de la Figure 51 illustre l'évaluation conjointe de l'expression mathématique $2+x+5$ par les systèmes Mathematica et MATLAB. Ce scénario suppose que Mathematica sous-traite à MATLAB l'évaluation de toute expression numérique, donc en particulier le calcul de la somme $2+5$. La méthode **getValue** de l'objet « $2+x+5$ » créé par

« Maple kernel » demande à l'objet « MATLAB kernel » la création d'une expression $2+5$ à l'aide du stimulus 15. Cette expression créée par MATLAB en possède la sémantique d'évaluation. L'appel 18 à la méthode **getValue** ne requiert donc pas le passage de l'objet « MATLAB kernel » en argument.

Le modèle structurel et comportemental des systèmes de calcul proposé offre une large palette de choix pour le codage des méthodes **getValue** des différentes sous-classes de la classe **Expression**. Lorsqu'une expression intermédiaire doit être construite, elle le sera de préférence à partir du système de calcul qui l'évaluera par la suite. Comme cela est le cas dans le diagramme de la Figure 51, cela évite de cloner ensuite l'expression avant de l'évaluer. Lorsqu'une expression intermédiaire **e** est disponible et doit être évaluée, l'appel **e.getValue()** l'évalue dans le système de calcul qui l'a créée, tandis qu'un appel **e.getValue(c)** offre le choix du système de calcul **c** pour l'évaluation de **e**. Un appel **c.value(e)** lui est préféré lorsque **c** est un système de calcul formel, et qu'un processus d'évaluation itératif est souhaité. Suivant chacun des choix opérés lors de l'implémentation des méthodes **getValue** pour la construction et l'évaluation des expressions intermédiaires, le système de calcul résultant sera plus ou moins formel et plus ou moins numérique, plus ou moins fiable et plus ou moins performant.

Le modèle structurel de système de calcul de la Figure 43, joint au modèle comportemental ébauché dans le diagramme de séquence de la Figure 51, permet de formaliser un processus d'évaluation d'une expression mathématique où peuvent alterner des étapes de transformation formelle et des étapes d'approximation numérique. Cela est déjà le propre des systèmes de calcul formel comme souligné en 1.1.1.2. Le modèle de coopération entre calcul formel et calcul numérique étend le modèle de système de calcul formel en permettant que chaque évaluation d'une expression soit au choix formelle ou numérique, que chaque étape de transformation formelle soit menée à bien par un système de calcul formel quelconque, et que chaque étape d'approximation numérique soit menée à bien par un système de calcul numérique quelconque.

Ce cadre conceptuel étant établi, surgissent plusieurs questions, directement liées à ces multiples possibilités de choix :

1. combien de systèmes de calcul utiliser ?
2. quelles expressions évaluer dans le cadre d'un système de calcul formel ?
3. quelles expressions évaluer dans le cadre d'un système de calcul numérique ?
4. quel(s) système(s) de calcul utiliser pour le calcul formel ?
5. quel(s) système(s) de calcul utiliser pour le calcul numérique ?

Le diagramme de classes de la Figure 43 tient compte des remarques faites en 2.1.1.2.1. Un système de calcul, formel ou numérique, référence certaines ou toutes les expressions qu'il a créées. Inversement, une expression référence le système de calcul à partir duquel elle a été créée. Nous suggérons qu'une expression donnée puisse référencer, non seulement le système de calcul qui l'a créée, mais aussi d'autres systèmes de calcul qui seront utilisés par la méthode **getValue** pour transformer formellement l'expression courante ou approcher numériquement sa valeur. Les systèmes de calcul référencés par une expression sont dès lors modélisés par une association de la classe **Expression** vers la classe **CalculationSystem**, dont le rôle **evaluators** a pour cardinalité 1..*, et non plus 1. Dans le cadre de cette présentation, sans perdre en généralité, nous limitons cette multiplicité à deux en choisissant de faire collaborer un système de calcul formel et un système de calcul numérique lors d'une simulation donnée. L'étude de la résolution d'équations continues, puis l'étude de la simulation de modèles continus par ce système de calcul symbolico-numérique, sont l'objet des parties suivantes. Elles complèteront le modèle de coopération proposé et donneront une réponse possible aux questions 2 et 3 relatives au choix du système de calcul le mieux adapté à chaque évaluation. Le Chapitre 3 propose une réalisation logicielle originale de système de calcul formel, nommée eXMSL, directement issue du modèle de coopération. Le Chapitre 4 constitue une réponse possible aux questions 4 et 5 : eXMSL et un système de calcul numérique particulier -la bibliothèque mathématique IMSL- y coopèrent pour simuler deux modèles continus.

2.2. Résolution symbolico-numérique d'équations continues

L'apport du calcul formel au sous-système de résolution numérique d'équations continues est étudié. La partie 2.2.1 envisage l'obtention d'expressions mathématiques formelles intervenant dans la résolution de modèles algébriques ou différentiels, puis leurs évaluations numériques fiables. La partie 2.2.2 modélise un système de résolution symbolico-numérique d'équations continues exploitant le principe de l'évaluation symbolico-numérique d'expressions mathématiques détaillé dans la partie 2.1. La partie 2.2.3 complète le modèle structurel par un scénario de résolution d'un système d'équations non linéaires à l'aide du système de résolution symbolico-numérique d'équations continues.

2.2.1. Obtention et évaluations numériques fiables d'expressions mathématiques formelles pour la résolution de …

2.2.1.1. … systèmes d'équations linéaires

Les systèmes linéaires à résoudre lors de la simulation numérique proviennent notamment de la linéarisation de modèles algébriques non linéaires lors de l'application de la méthode de Newton-Raphson, de l'approximation des dérivées partielles par des formules aux différences finies, ou de la recherche des coordonnées d'une fonction solution dans un espace de fonctions polynômes par morceaux lors de l'application de la méthode des éléments finis. La résolution numérique fiable et performante de systèmes d'équations linéaires est donc un pré-requis à l'application de la plupart des méthodes du calcul numérique. Les systèmes linéaires issus de la modélisation de systèmes couplant divers phénomènes physiques peuvent être de grande taille, c'est-à-dire comprendre plusieurs centaines de milliers d'inconnues, voire des millions. Selon (Scott 2001) la résolution répétée de tels systèmes linéaires, creux et très fortement non symétriques, *« est en général l'étape la plus coûteuse en temps calcul lors de la simulation de procédés chimiques de grande taille, requérant souvent plus de quatre-vingt dix pour cent du temps d'exécution total »*. Afin de résoudre efficacement les systèmes linéaires creux, des techniques de gestion de la structure creuse des matrices sont couplées à des algorithmes variés de résolution exclusivement numériques afin de concilier le temps de calcul et l'exactitude du résultat.

Lorsqu'une méthode directe est utilisée pour résoudre un grand système creux, les techniques de renumérotation des équations peuvent aider à limiter le phénomène de remplissage. Ces techniques sont en fait très proches de certaines techniques du calcul formel : les méthodes d'arrangement travaillent sur des listes d'équations et de variables, tandis que les

systèmes de calcul formel transforment des expressions, vues comme des listes de sous-expressions.

Lorsqu'une méthode itérative est utilisée, et tout particulièrement lorsqu'une méthode *« matrix-free »* telle que GMRES (Saad & Schultz 1986) est employée, la méthode numérique peut tirer avantage d'une représentation symbolico-numérique du système linéaire à résoudre $A \cdot x = b$. $A \cdot x$ est alors représenté comme une liste d'expressions algébriques formelles, résultat du produit de la matrice réelle A par le vecteur de symboles $x = \{x_1, x_2, ..., x_n\}$. Ainsi, chaque produit de la matrice incidente A par un vecteur de nombres réels r est obtenu efficacement en affectant la valeur x à r, puis en évaluant numériquement l'expression formelle $A \cdot x$. Le gain en performance provient de l'absence d'une structure de données et d'un algorithme dédiés au stockage et à la manipulation des éléments non nuls de la matrice A. La structure de données stockant le résultat du produit $A \cdot x$ est construite sur le même modèle, ou type de données, que celui de toutes les autres expressions mathématiques manipulées par le système de calcul formel. L'algorithme d'évaluation de cette expression est celui permettant d'évaluer toutes les autres expressions mathématiques manipulées par le système de calcul formel.

Le modèle de système de résolution symbolico-numérique d'équations continues proposé n'établit pas de distinction entre les systèmes linéaires et les systèmes non linéaires. Une méthode unique **nonLinearEquationRoot**[15], de recherche d'une solution réelle, est proposée dans tous les cas. Les arguments de la méthode **nonLinearEquationRoot** sont deux expressions : l'équation, ou la liste d'équations à résoudre, et le symbole ou la liste de symboles désignant les inconnues. Ces deux familles de systèmes d'équations sont traitées à l'aide des méthodes appliquées aux systèmes d'équations non linéaires, en particulier la méthode de Newton-Raphson.

2.2.1.2. … systèmes d'équations et d'inéquations non linéaires

Pour résoudre un système d'équations non linéaires à l'aide des outils proposés par le calcul numérique, une transposition est systématiquement requise. La formulation déclarative, qui est celle adoptée par l'ingénieur métier ou le chercheur lorsqu'ils écrivent les équations représentant

[15] Le modèle structurel de système de résolution symbolico-numérique d'équations continues proposé suit une syntaxe proche de celle du système de calcul formel Mathematica. La correspondance n'est cependant pas exacte : Mathematica distingue par exemple la résolution numérique de systèmes d'équations linéaires (**`LinearSolve`**) de la résolution numérique de systèmes d'équations non linéaires (**`FindRoot`**).

le système étudié sur le papier, est remplacée par une formulation fonctionnelle, requise par les langages procéduraux dans lesquels sont écrits la majorité des algorithmes de résolution.

A l'inverse, les systèmes de calcul formel s'accommodent d'une transcription littérale des équations. Dans le cas d'une résolution numérique d'équations non linéaires, la valeur $F(x)$ de la fonction résidu F au point inconnu x est calculée automatiquement, en soustrayant les membres droits des membres gauches des équations. Lorsque la méthode de Newton-Raphson est appliquée, la matrice Jacobienne $F'(x)$ évaluée au point x est obtenue par dérivation formelle de la fonction résidu. A chaque itération de Newton, k, les expressions formelles $F(x)$ et $F'(x)$ sont évaluées numériquement après avoir assigné à $x = \{x_1, x_2, ..., x_n\}$ les valeurs réelles $\{x_{k,1}, x_{k,2}, ..., x_{k,n}\}$.

Le système de résolution d'équations non linéaires présenté combine des caractéristiques propres aux systèmes de calcul formel et des caractéristiques propres au calcul numérique.

Ce système de résolution d'équations non linéaires est symbolique car, étant donné un symbole $x = \{x_1, x_2, ..., x_n\}$:

1. le système d'équations non linéaires est formulé de manière déclarative à l'aide de nombres, de symboles, d'opérateurs arithmétiques et des fonctions mathématiques usuelles ;
2. le résultat $F(x)$ de l'application de la fonction résidu F au point x est calculé formellement ;
3. la matrice Jacobienne $F'(x)$ évaluée au point x est calculée formellement.

Ce système de résolution d'équations non linéaires est numérique car :

4. la mise en œuvre de l'algorithme de résolution est un code informatique existant, choisi parmi une des nombreuses bibliothèques mathématiques numériques éprouvées et performantes. Il s'agit la plupart du temps d'une mise en œuvre de la méthode de Newton-Raphson ou de méthodes quasi-Newton. Le texte source du code informatique utilisé n'est pas nécessairement disponible, l'interface d'appel est par contre connue précisément ;
5. à la demande du code de résolution choisi, l'expression formelle $F(x)$ est évaluée numériquement pour diverses valeurs numériques réelles $\{x_{k,1}, x_{k,2}, ..., x_{k,n}\}$ de x ;
6. à la demande du code de résolution choisi, l'expression formelle $F'(x)$ est évaluée numériquement pour diverses valeurs numériques réelles $\{x_{k,1}, x_{k,2}, ..., x_{k,n}\}$ de x.

Le principe de résolution numérique améliorée par le calcul formel introduit en 1.1.4.3 est proposé par certains systèmes de calcul formel. Maple à travers la fonction **fsolve**, ou Mathematica à travers la fonction **FindRoot**, disposent déjà d'algorithmes de résolution symbolico-numérique vérifiant les points 1, 2, 3, 5 et 6. La réutilisation systématique de codes de résolution existants n'est cependant pas envisagée, hormis l'intégration de la bibliothèque mathématique Nag dans Maple décrite dans (Trefethen & Ford 2000). Le système de résolution

symbolico-numérique d'équations continues décrit dans la partie 2.1 fait de cette réutilisation un principe pour atteindre les objectifs de fiabilité et de performance.

Le mode d'évaluation numérique adopté pour les points 5 et 6 diffère à la fois de celui des systèmes de calcul formel, qui disposent d'une arithmétique rationnelle exacte (cf. 1.1.2.1) et d'une arithmétique réelle en précision multiple (cf. 1.1.2.2), et de celui des bibliothèques mathématiques numériques qui calculent dans le cadre de l'arithmétique réelle en virgule flottante. Afin de concilier les objectifs de fiabilité et de performance nous choisissons de représenter tous les nombres à l'aide de structures de données élémentaires et statiques : les mots machine de l'architecture informatique utilisée. Plus précisément, l'évaluation d'expressions mathématiques où figurent des nombres, des constantes numériques prédéfinies, des opérateurs arithmétiques et les fonctions mathématiques usuelles suit les règles suivantes :

- chaque fois que cela a été prévu a priori, le résultat exact de l'application d'une fonction mathématique à un argument numérique est retourné. A titre d'exemple, l'évaluation numérique de $\log(e)$ retourne le nombre entier 1, et l'évaluation numérique de $\cos(\pi)$ retourne le nombre entier -1 ;
- le système adopte une arithmétique rationnelle exacte si et seulement si les numérateurs et les dénominateurs des opérandes rationnels et du résultat rationnel de l'opération arithmétique peuvent tous être représentés par des nombres entiers en machine ;
- l'arithmétique réelle est l'arithmétique réelle en virgule flottante ;
- le résultat de l'évaluation numérique est systématiquement converti en un nombre réel en virgule flottante, en un vecteur de nombres réels en virgule flottante, ou en une matrice de nombres réels en virgule flottante.

Le dernier point est presque imposé par la réutilisation de bibliothèques mathématiques numériques où le nombre réel en virgule flottante est le type de données de prédilection. Si l'objectif de fiabilité du processus de résolution prime sur l'objectif de performance, il serait intéressant d'effectuer les calculs intermédiaires dans le cadre d'une arithmétique rationnelle exacte et d'une arithmétique réelle à précision multiple, voire exacte (cf. 1.1.2.4).

Le système de résolution symbolico-numérique d'équations non linéaires ci-dessus prend en charge deux généralisations qui nous ont parues importantes pour aborder la simulation de systèmes complexes :

1. les systèmes d'équations non linéaires sous-contraints ou sur-contraints ;
2. les systèmes d'équations et d'inéquations non linéaires.

Dans le cadre de stratégies de simulation où les modèles de plusieurs sous-systèmes constituant un système complexe sont résolus indépendamment, les systèmes d'équations non linéaires associés à chaque sous-système physique ne comportent pas nécessairement autant d'équations que d'inconnues. Les systèmes d'équations non linéaires sous contraints, i.e. où le nombre d'équations est strictement inférieur au nombre d'inconnues, ou sur-contraints, i.e. où le nombre d'équations est strictement supérieur au nombre d'inconnues, peuvent être résolus par la méthode de Newton-Raphson sous réserve de généraliser la définition de l'opérateur de Newton. Le schéma de Newton pour calculer l'itéré x_{k+1} à partir de l'itéré précédent x_k s'écrit :

$$x_{k+1} = x_k - \mathrm{N}(F)(x_k) \cdot F(x_k)$$

Équation 3 – Schéma itératif de Newton.

Dans le cas où le nombre d'équations est égal au nombre d'inconnues, l'opérateur de Newton de la fonction F évalué au point x_k s'écrit :

$$\mathrm{N}(F)(x_k) = \left[F'(x_k)\right]^{-1}$$

Équation 4 – Opérateur de Newton lorsque le nombre d'équations est égal au nombre d'inconnues.

Si le nombre d'inconnues est supérieur strictement au nombre d'équations, selon (Dedieu 2006) le schéma précédent s'applique avec :

$$\mathrm{N}(F)(x_k) = \left[F'(x_k)\right]^t \cdot \left[F'(x_k) \cdot F'(x_k)^t\right]^{-1}$$

Équation 5 – Opérateur de Newton lorsque le nombre d'inconnues est supérieur strictement au nombre d'équations.

Si le nombre d'inconnues est inférieur strictement au nombre d'équations, selon (Dedieu 2006) le schéma précédent s'applique avec :

$$\mathrm{N}(F)(x_k) = \left[F'(x_k) \cdot F'(x_k)^t\right]^{-1} \cdot \left[F'(x_k)\right]^t$$

Équation 6 – Opérateur de Newton lorsque le nombre d'inconnues est inférieur strictement au nombre d'équations.

Équation 3 est donc une formulation générale de l'algorithme de Newton-Raphson, valide à la fois pour des systèmes non linéaires saturés, sous-contraints ou sur-contraints. Cependant les codes de résolution numérique de systèmes d'équations non linéaires imposent la plupart du

temps que le nombre d'équations soit égal au nombre d'inconnues. Une première généralisation du système de résolution symbolico-numérique d'équations non linéaires choisit de formuler la résolution de systèmes d'équations non linéaires par l'algorithme de Newton-Raphson comme une résolution d'équations différentielles ordinaires du premier ordre (Hirsch & Smale 1979). Le schéma itératif Équation 3 est remplacé par la formulation différentielle Équation 7.

$$\boxed{\forall t \in \Re^{+}; x'(t) = -N(F)(x(t)) \cdot F(x(t))}$$

Équation 7 – Formulation différentielle du schéma de Newton.

La résolution d'un système d'équations non linéaires consiste alors, partant d'une estimation de la solution $x_0 = x(0)$, à intégrer le système d'équations différentielles ordinaires Équation 7 . Dans le cas d'un système algébrique saturé ou sous-contraint, il existe la plupart du temps une valeur t^* du paramètre d'intégration telle que $F(x(t^*)) = 0$. L'intégration s'arrête en ce point. Dans le cas d'un système algébrique sur-contraint, l'opérateur de Newton donné en Équation 6 correspond à la résolution d'un problème de moindres carrés : il s'agit de trouver x^* minimisant la norme du vecteur $F(x)$. Dans ce cas, l'intégration de Équation 7 s'arrête pour une valeur t^* du paramètre d'intégration telle que $x'(t^*) = 0$. Lorsque le nombre d'équations est égal au nombre d'inconnues, la formulation algébrique du schéma de Newton est appliquée, suivie de la formulation différentielle lorsque la formulation algébrique échoue.

La seconde généralisation du système de résolution symbolico-numérique d'équations non linéaires consiste à traiter les systèmes constitués non seulement d'équations mais aussi d'inéquations. Tout comme dans le cas d'un système d'équations non linéaires, la formulation d'un système d'équations et d'inéquations non linéaires est formelle. Une première transformation formelle appliquée à ce système produit un système d'équations non linéaires, ce dernier est ensuite traité comme décrit précédemment. Sur le modèle de l'algorithme du simplexe, la transformation formelle initiale ajoute des variables d'écart au modèle. Contrairement à l'algorithme du simplexe, le carré des variables d'écart apparaît dans les équations issues des inéquations transformées. A titre d'exemple, le modèle suivant :

$$\forall (x, y) \in \Re^2 \begin{cases} x^2 + y^2 = 1 \\ y = (x - 0.3)^2 \\ x \cdot y \geq 0 \end{cases}$$

est transformé en introduisant la variable d'écart α :

$$\forall (x, y, \alpha) \in \Re^2 \begin{cases} x^2 + y^2 = 1 \\ y = (x - 0.3)^2 \\ x \cdot y = \alpha^2 \end{cases}$$

Il faut noter que cette généralisation est liée à la précédente : le système d'équations résultant de l'introduction de variables d'écart n'est pas nécessairement carré, il peut être sous-contraint ou sur-contraint.

2.2.1.3. … systèmes d'équations algébro-différentielles

En adoptant les notations de Claude Gomez dans son cours sur les systèmes dynamiques (Gomez 2005), la forme la plus générale d'un système algébro-différentiel est celle du système implicite $\begin{cases} F(t, y(t), y'(t)) = 0 \\ y(t_0) = y_0 \end{cases}$, où $F : U \subset I \times \Re^n \times \Re^n \to \Re^n$, U est un ouvert, I est un intervalle de $\Re$ et $(t_0, y_0) \in I \times \Re^n$. Les dérivées partielles de la fonction résidu F sont représentées en adoptant la notation utilisée par Jerrold E. Marsden[16], dans (Wendlandt & Marsden 1997) notamment.

Sur le principe du système de résolution symbolico-numérique d'équations non linéaires introduit précédemment, le système de résolution d'équations algébro-différentielles présenté combine des caractéristiques propres aux systèmes de calcul formel, et des caractéristiques propres au calcul numérique.

Ce système de résolution d'équations algébro-différentielles est symbolique car, étant donnés un symbole t, un symbole $y = \{y_1, y_2, ..., y_n\}$ et un symbole $dy = \{dy_1, dy_2, ..., dy_n\}$:

1. le système d'équations algébro-différentielles est formulé de manière déclarative ;
2. le résultat $F(t, y, dy)$ de l'application de la fonction résidu F au point (t, y, dy) est calculé formellement ;

---

[16] A titre d'exemple, étant donnée une fonction F définie par $F : \begin{array}{l} \Re^m \times \Re^n \times \Re^o \to \Re^p \\ (x, y, z) \mapsto F(x, y, z) \end{array}$, la notation $D_1 F(x, y, z)$ désigne l'évaluation de la fonction dérivée partielle $D_1 F$, par rapport aux m premières variables, au point (x, y, z). Cette notation se veut plus stricte que la notation usuelle $\frac{\partial F}{\partial x}(x, y, z)$, qui utilise un même symbole pour les variables de dérivation partielle et le point d'évaluation.

3. les dérivées partielles $D_2F(t,y,dy)$, de la fonction résidu F par rapport aux variables dépendantes, évaluées au point (t,y,dy), sont calculées formellement ;
4. les dérivées partielles $D_3F(t,y,dy)$, de la fonction résidu F par rapport aux dérivées de des variables dépendantes, évaluées au point (t,y,dy), sont calculées formellement.

Ce système de résolution d'équations algébro-différentielles est numérique car :

5. la mise en œuvre de l'algorithme de résolution est un code informatique existant, choisi parmi une des nombreuses bibliothèques mathématiques numériques éprouvées et performantes. Le texte source du code informatique utilisé n'est pas nécessairement disponible, l'interface d'appel est par contre connue précisément ;
6. à la demande du code de résolution choisi, l'expression formelle $F(t,y,dy)$ est évaluée numériquement pour diverses valeurs numériques réelles $\left(t_k, y_{k,1},...,y_{k,n}, dy_{k,1},...,dy_{k,n}\right)$ de (t,y,dy) ;
7. à la demande du code de résolution choisi, l'expression formelle $D_2F(t,y,dy)$ est évaluée numériquement pour diverses valeurs numériques réelles $\left(t_k, y_{k,1},...,y_{k,n}, dy_{k,1},...,dy_{k,n}\right)$ de (t,y,dy) ;
8. à la demande du code de résolution choisi, l'expression formelle $D_3F(t,y,dy)$ est évaluée numériquement pour diverses valeurs numériques réelles $\left(t_k, y_{k,1},...,y_{k,n}, dy_{k,1},...,dy_{k,n}\right)$ de (t,y,dy).

Le système de résolution symbolico-numérique d'équations algébro-différentielles prend en charge le calcul de conditions initiales cohérentes. Plusieurs travaux de la littérature abordent ce sujet, primordial car les méthodes de résolution d'équations algébro-différentielles de type prédicteur-correcteur requièrent des conditions initiales cohérentes, ou quasi cohérentes, portant sur les variables dépendantes et leurs dérivées d'ordre un (Brenan, Campbell, & Petzold 1996).

(Brown, Hindmarsh, & Petzold 1998) recense différentes approches adoptées pour aborder ce problème. Une méthode strictement numérique, due à (Berzins, Dew, & Furzeland 1989), intègre le système algébro-différentiel sur un horizon de temps réduit, puis procède à une intégration inverse ; seules des conditions initiales quasi cohérentes sont traitées. (Rascol et al. 1998) illustre une application de ce calcul, dont les détails sont analysés dans (Le Lann 1999). Des méthodes basées sur la théorie des graphes, telle que celle présentée dans (Pantelides 1988), procèdent à une analyse structurelle des équations et des variables dépendantes afin de déterminer un système d'équations non linéaires caractérisant les conditions initiales cohérentes. (Campbell 1986) suggère l'utilisation de développements en séries de Taylor, où les dérivées de la matrice Jacobienne sont obtenues par différentiation automatique. D'autres auteurs reprennent ce principe avec une approximation numérique des dérivées.

Par ailleurs, des travaux plus théoriques envisagent de déterminer des conditions d'existence et d'unicité des solutions de classes d'équations algébro-différentielles. Bien qu'ardues, ces approches robustes sont d'un intérêt pratique certain. (Campbell & Griepentrog 1995) donne des conditions suffisantes de solvabilité, vérifiables en utilisant des systèmes de calcul numérique ou formel. (Kunkel & Mehrmann 1998) s'appuie sur des résultats théoriques pour construire des méthodes numériques de détermination de conditions initiales cohérentes et calculer des solutions régulières.

Avec un objectif plus modeste, le système de résolution symbolico-numérique proposé, traite la question des conditions initiales cohérentes pour certains systèmes d'équations algébro-différentielles d'index un selon une démarche simple et systématique :

1. toutes les équations algébriques sont dérivées formellement, puis évaluées pour la valeur initiale de la variable indépendante ;
2. le système d'équations non linéaires constitué par l'union des équations différentielles, des équations algébriques, et des équations algébriques différentiées, évaluées pour la valeur initiale de la variable indépendante, est résolu pour produire des conditions initiales cohérentes.

La première étape tire avantage du fait que le système de calcul proposé fournit des dérivées analytiques. La seconde étape est permise par le fait que le système de calcul proposé prend en charge des systèmes d'équations non linéaires structurellement sur-contraints, mais en fait saturés si le problème est bien posé. Ce traitement suppose que la seule dérivation des équations algébriques produit un ensemble de contraintes suffisantes pour déterminer complètement des conditions initiales cohérentes. La classe des systèmes d'équations algébro-différentielles traitée est donc un sous-ensemble des systèmes dont l'index de différentiation vaut un.

2.2.1.4. … problèmes d'optimisation sous contraintes non linéaires

La programmation non linéaire consiste à calculer la valeur numérique de variables minimisant un critère, ces variables vérifiant des contraintes non linéaires. Plus précisément, étant donnés f, $(g_i)_{i\in\{1,\dots,u\}}$ et $(h_j)_{j\in\{1,\dots,v\}}$ des fonctions définies de $\Re^n$ dans $\Re$, et $\mathrm{E}=\{x\in\Re^n;\forall i\in\{1,\dots,u\};\forall j\in\{1,\dots,v\};g_i(x)=0\wedge h_j(x)\geq 0\}$, un problème de minimisation sous contraintes non linéaires consiste à trouver $x^*\in\Re^n$ tel que $x^*=\min_{x\in\mathrm{E}} f(x)$. Le travail présenté suppose le critère et les contraintes différentiables. Un minimum local est recherché à l'aide d'une méthode de programmation quadratique successive utilisant les dérivées des fonctions f, $(g_i)_{i\in\{1,\dots,u\}}$ et $(h_j)_{j\in\{1,\dots,v\}}$.

Sur le principe des systèmes de résolution symbolico-numérique d'équations algébriques et/ou différentielles introduits précédemment, le système de résolution de problèmes de programmation non linéaire présenté combine des caractéristiques propres aux systèmes de calcul formel, et des caractéristiques propres au calcul numérique.

Ce système de résolution de problèmes de programmation non linéaire est symbolique car, étant donné un symbole $x = \{x_1, x_2, ..., x_n\}$:

1. le problème de programmation non linéaire est formulé de manière déclarative, en particulier les contraintes ;
2. le résultat $f(x)$ de l'application de la fonction critère au point x est calculé formellement ;
3. les résultats $(g_i(x))_{i\in\{1,...,u\}}$ de l'application des fonctions contraintes égalités $(g_i)_{i\in\{1,...,u\}}$ au point x sont calculés formellement ;
4. les résultats $(h_j(x))_{j\in\{1,...,v\}}$ de l'application des fonctions contraintes inégalités $(h_j)_{j\in\{1,...,v\}}$ au point x sont calculés formellement ;
5. la dérivée $f'(x)$ de la fonction critère f au point x est calculée formellement ;
6. les dérivées $(g_i'(x))_{i\in\{1,...,u\}}$ des fonctions contraintes égalités $(g_i)_{i\in\{1,...,u\}}$ au point x sont calculées formellement ;
7. les dérivées $(h_j'(x))_{j\in\{1,...,v\}}$ des fonctions contraintes inégalités $(h_j)_{j\in\{1,...,v\}}$ au point x sont calculées formellement.

Ce système de résolution d'équations algébro-différentielles est numérique car :

8. la mise en œuvre de l'algorithme de résolution est un code informatique existant, choisi parmi une des nombreuses bibliothèques mathématiques numériques éprouvées et performantes. Il s'agit la plupart du temps d'une mise en œuvre de la méthode de Gear (Gear 1971). Le texte source du code informatique utilisé n'est pas nécessairement disponible, l'interface d'appel est par contre connue précisément ;
9. à la demande du code de résolution choisi, les expressions formelles $f(x)$, $(g_i(x))_{i\in\{1,...,u\}}$, $(h_j(x))_{j\in\{1,...,v\}}$ sont évaluées numériquement pour diverses valeurs numériques réelles $(x_{k,1},...,x_{k,n})$ de x ;
10. à la demande du code de résolution choisi, les expressions formelles $f'(x)$, $(g_i'(x))_{i\in\{1,...,u\}}$, $(h_j'(x))_{j\in\{1,...,v\}}$ sont évaluées numériquement pour diverses valeurs numériques réelles $(x_{k,1},...,x_{k,n})$ de x.

2.2.2. Modèle structurel d'un système de résolution symbolico-numérique d'équations continues

Comme l'a montrée la partie 2.2.1, la contribution proposée du calcul formel à la résolution de systèmes d'équations continues consiste en l'obtention préalable, par un système de

calcul formel, d'expressions mathématiques formelles qui, par la suite, seront évaluées numériquement à la demande d'un système de calcul numérique. Le système de calcul proposé est donc nécessairement un système de calcul formel amené à coopérer avec un système de calcul numérique existant. La classe **SymbolicNumericalCalculationSystem** du paquetage **computer-algebra-systems** modélise un système de calcul apte à appliquer cette stratégie de coopération de type complémentaire.

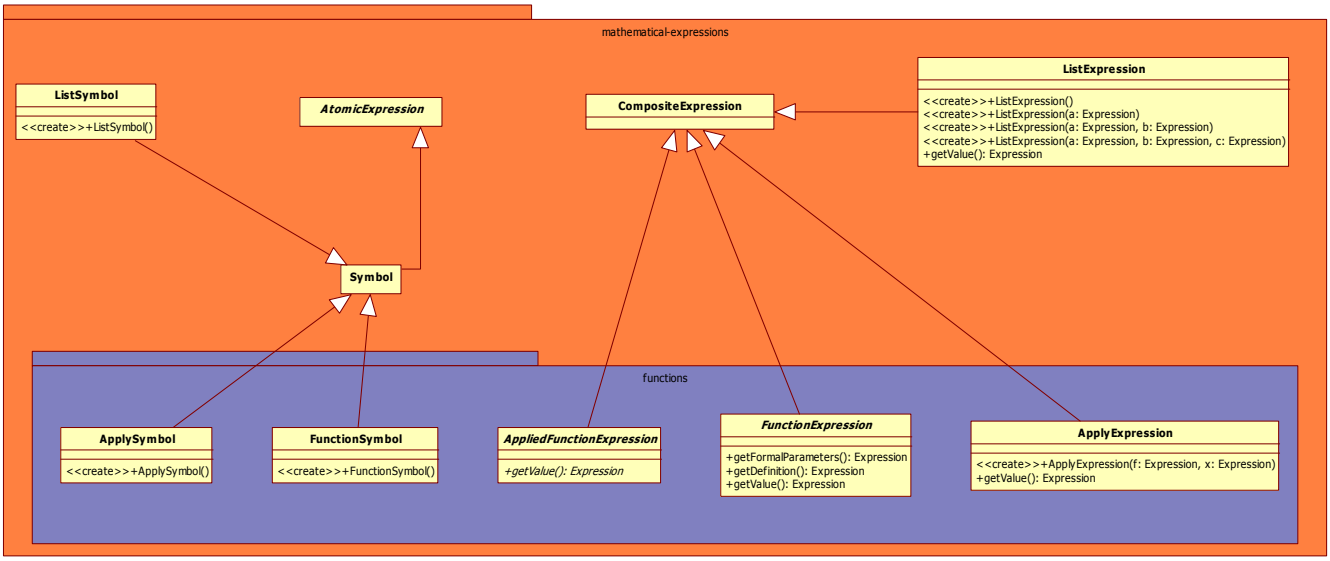

Figure 52 – Modèle structurel pour la définition de fonctions dans un système de calcul.

Avant de détailler l'interface de la classe **SymbolicNumericalCalculationSystem**, il est nécessaire de modéliser les services de résolution de systèmes d'équations continues présents dans les systèmes de calcul numérique ou formel, et ce pour les quatre familles de problèmes évoqués dans la partie 2.2.1. Cette modélisation impose de formaliser la notion de fonction au sens mathématique car cette notion est omniprésente lors de la mise en œuvre des méthodes de résolution du calcul numérique. Comme le montre le diagramme de classes de la Figure 52, une fonction est vue ici comme une expression composite dont les deux sous-expressions sont respectivement le paramètre formel ou la liste ordonnée des paramètres formels, et la définition de la fonction. Cette définition est éventuellement une liste ordonnée de définitions dans le cas de la représentation de multiples fonctions coordonnées. La classe **FunctionExpression** représente une expression –**definition**- dans laquelle peuvent apparaître les symboles **formalParameters**. L'évaluation d'un objet du type **FunctionExpression** par la méthode **getValue** retourne l'objet lui-même. Une fonction peut notamment être appliquée à une liste **arguments** pour produire une nouvelle expression du type **AppliedFunctionExpression**. L'évaluation d'un objet de ce type par la méthode **getValue** calcule l'image de la fonction au point argument.

Dans un système de calcul numérique, la définition d'une fonction est une séquence de plusieurs instructions dans lesquelles figurent les symboles **formalParameters** en tant que variables

typées. Les types, implicites ou explicites, des symboles **formalParameters** sont numériques et imposent donc que le point d'évaluation **arguments** soit une liste ordonnée d'expressions numériques. Selon le modèle structurel des expressions mathématiques choisi dans la Figure 34, chaque argument est nécessairement un objet d'un sous-type de **Number**. L'évaluation de la fonction au point **arguments** consiste alors à exécuter successivement des instructions d'affectation des éléments de **arguments** aux éléments de **formalParameters**, puis la séquence **definition**.

Dans un système de calcul formel, la définition d'une fonction est également une séquence de plusieurs instructions dans lesquelles figurent les symboles **formalParameters**. Contrairement aux variables des fonctions du calcul numérique, ces symboles ne contraignent pas les types des éléments de **arguments**. Selon le modèle structurel des expressions mathématiques choisi dans la Figure 34, chaque argument est un objet d'un sous-type de **Expression**. L'évaluation de la fonction au point **arguments** consiste à remplacer dans **definition** chaque occurrence d'un symbole de la liste ordonnée **formalParameters** par l'élément de même rang dans **arguments**, puis à évaluer l'expression ainsi obtenue.

L'absence d'une sémantique commune pour l'évaluation d'une fonction en un point explique le fait que la classe **AppliedFunctionExpression** du diagramme de classes de la Figure 52 soit abstraite. Un comportement par défaut de la méthode **getValue** de cette classe ne peut pas être exhibé. Malgré les sémantiques différentes associées à l'application d'une fonction en un point, dans les systèmes de calcul formel et de calcul numérique, le modèle structurel des expressions mathématiques adopté offre un socle commun à partir duquel est modélisée la notion de fonction dans un système de calcul. Les méthodes **list** et **apply** de la classe **CalculationSystem**, enrichie dans le diagramme de classes de la Figure 53, complètent cette déclaration. **list** construit une liste ordonnée d'éléments, tandis que **apply** applique une expression à une liste d'arguments. L'exécution de **apply(f, list(x, y))** consiste à construire une expression dont la tête est **f** et dont les sous-expressions sont **x** et **y**. Dans un système de calcul numérique **f** représente nécessairement une fonction au sens informatique du terme, et l'expression retournée par la méthode **apply** a alors la signification particulière de l'appel **f(x, y)** à **f** avec des arguments d'appel **x** et **y**. Dans un système de calcul formel la sémantique liée à l'expression retournée par la méthode **apply** reste générale.

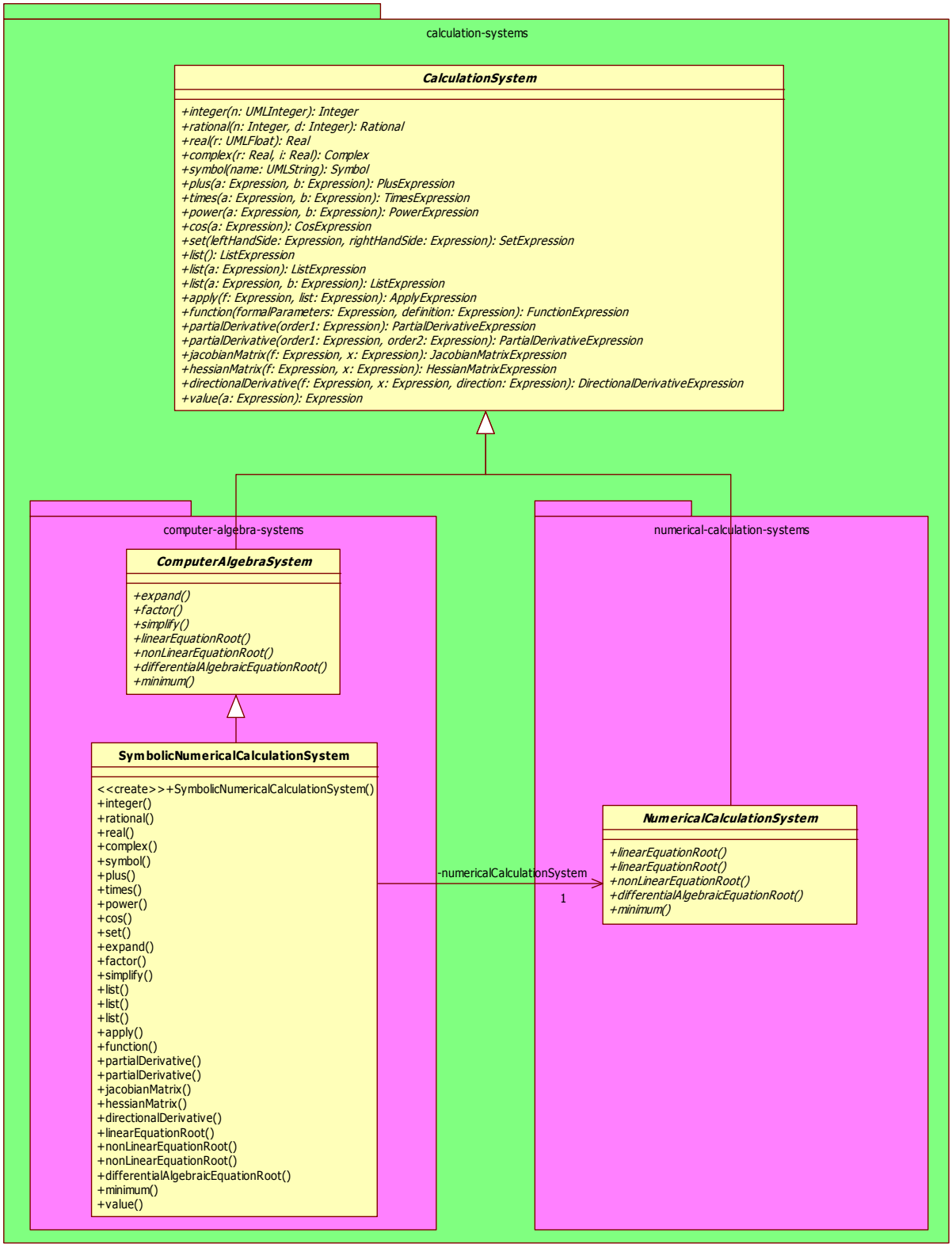

Figure 53 – Modèle structurel du système de calcul symbolico-numérique proposé.

A partir du modèle de fonction introduit ci-dessus sont construits les représentations des outils nécessaires au calcul différentiel élémentaire. L'ensemble des déclarations de méthodes pour produire des fonctions dérivées est regroupé dans la classe **CalculationSystem**. Il peut être surprenant de voir apparaître l'opérateur linéaire de dérivation partielle par rapport à une variable donnée, modélisé par la méthode **partialDerivative**, dans l'interface de la classe **CalculationSystem**. Pourtant une bibliothèque mathématique pour le calcul par différences finies, ou un outil de

différentiation automatique, produisent tous deux, à partir d'une séquence d'instructions représentant une fonction mathématique, une autre expression mathématique représentant la dérivée partielle de la fonction par rapport à une variable donnée, c'est-à-dire une nouvelle fonction. Ces systèmes de calcul numérique proposent tous deux un accès à l'opérateur linéaire de dérivation partielle par rapport à une variable donnée, bien que le résultat de l'application de cet opérateur à une fonction soit un programme et non une structure de données comme cela est le cas dans les systèmes de calcul formel. Les arguments d'appel de la méthode **partialDerivative** sont uniquement les ordres de dérivation par rapport aux variables successives de la fonction à laquelle sera appliqué l'opérateur de dérivation partielle. Dès lors, étant donnée une fonction **f** de deux variables, l'accès à la dérivée partielle de **f** par rapport à la première variable uniquement, $\partial^{(1,0)} f$, prend la forme suivante :

```
c.apply(c.partialDerivative(integer(1), integer(0)), f)
```

L'accès à la valeur de la dérivée partielle de **f** par rapport à la première variable uniquement au point *(x, y)*, $\partial^{(1,0)} f(x, y)$, prend la forme suivante :

```
c.apply(c.apply(c.partialDerivative(integer(1), integer(0)), f), list(x, y))
```

Les méthodes **jacobianMatrix** et **hessianMatrix** produisent des expressions représentant respectivement la dérivée première et la dérivée seconde d'une fonction **f** appliquée en un point **x**. La définition de ces méthodes dans la classe **SymbolicNumericalCalculationSystem** s'appuie sur les appels à la méthode **partialDerivative**. Si la méthode **jacobianMatrix** ne fait aucune hypothèse sur les dimensions des espaces de départ et d'arrivée de la fonction donnée en argument, **hessianMatrix** impose une fonction à valeurs dans un espace de dimension un. La dérivée directionnelle d'une fonction *f* en un point *x* selon une direction donnée **direction**, i.e. le produit scalaire de la matrice Jacobienne de *f* au point *x* par le vecteur **direction**, s'exprime à l'aide de la méthode **directionalDerivative**. Les définitions des méthodes **partialDerivative**, **jacobianMatrix**, **hessianMatrix** et **directionalDerivative** dans la classe **SymbolicNumericalCalculationSystem** appliquent les règles de dérivation formelle à des expressions symbolico-numériques comme le font les autres systèmes de calcul formel. L'expression analytique des dérivées produites semble donc exacte. En fait le calcul numérique approché, éventuellement en précision multiple, effectué sur les termes numériques conduit à une évaluation approchée et non exacte des dérivées. L'approximation fonctionnelle des dérivées produites par le système de calcul formel s'avèrera plus fiable que celle obtenue par une formule aux différences finies.

Une fois établi un modèle structurel des fonctions mathématiques (différentiables) et de leurs dérivées, une description des services de résolution des systèmes d'équations continues dans les environnements de calcul devient possible. L'interface de ces services est foncièrement différente entre les systèmes de calcul formel et les systèmes de calcul numérique. Si le calcul formel accepte une spécification du problème à résoudre à l'aide d'équations et d'inéquations, le calcul numérique exige une transposition du problème initial sous une forme fonctionnelle adaptée aux algorithmes du calcul numérique, algorithmes basés le plus souvent sur l'évaluation répétée de fonctions et de leurs dérivées. Dès lors, une factorisation des méthodes de résolution de systèmes d'équations non linéaires, algébro-différentielles ou des méthodes d'optimisation dans la classe **CalculationSystem** est impossible. Le modèle structurel de la Figure 53 organise ces méthodes de résolution dans les classes abstraites **NumericalCalculationSystem** et **ComputerAlgebraSystem**. Le diagramme de classes de la Figure 43 se voit désormais enrichi de services de résolution de problèmes continus, services distribués entre calcul formel et calcul numérique. La Figure 54 précise les signatures des méthodes de résolution présentes dans les classes **NumericalCalculationSystem** et **ComputerAlgebraSystem,** et en souligne les différences.

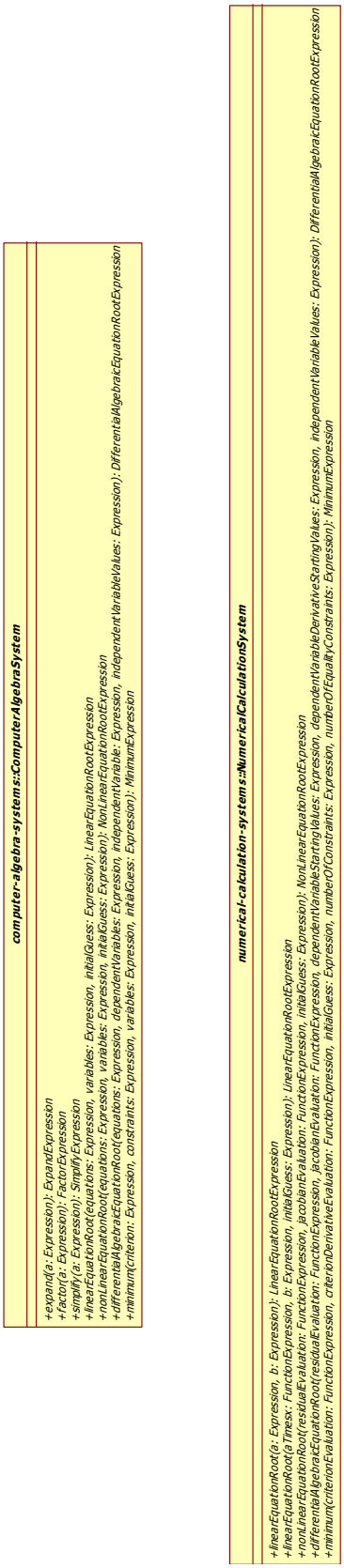

Figure 54 – Interfaces déclarées par un système de calcul formel et un système de calcul numérique.

En général les systèmes de calcul, formels ou numériques, présentent plusieurs services de résolution d'un système d'équations linéaires. Ceux-ci diffèrent par le fait que l'interface de la méthode de résolution exige entre autres :

- la fourniture de la matrice incidente du système ou non ;

- une topologie de matrice particulière ou non ;
- des caractéristiques spectrales de la matrice incidente ou non ;
- une solution approchée ou non.

Le modèle structurel de système de calcul numérique présenté se limite à la description de deux services de résolution de systèmes d'équations linéaires. Le premier correspond à la méthode **linearEquationRoot** de la classe **NumericalCalculationSystem** dont les paramètres formels sont la matrice incidente **a** et le vecteur constant **b**. Cette signature correspond plutôt à une résolution par une méthode directe, sans prise en compte particulière du profil de la matrice incidente. Une deuxième méthode **linearEquationRoot**, proposée dans la classe **NumericalCalculationSystem**, fait le choix de ne pas stocker la matrice incidente et d'exiger une estimation de la racine du système d'équations. La méthode de résolution mise en œuvre dans les classes héritières de la classe **NumericalCalculationSystem** est alors une méthode de résolution itérative, dite *« matrix-free »* car le profil de la matrice incidente n'est pas conservé. L'algorithme de résolution appelle la fonction **aTimesx** pour différentes valeurs du vecteur des variables, initialisé à **initialGuess**.

La signature de la méthode **linearEquationRoot** de la classe **ComputerAlgebraSystem** ne présuppose en rien la méthode de résolution du système d'équations linéaires. Ce dernier, fourni comme une liste **equations** où apparaissent chacun des symboles de la liste **variables**, est transcrit sans transposition préalable. La fourniture brute des équations sous la forme d'expressions mathématiques évite un stockage initial de multiples zéros, dans le cas où la matrice incidente est creuse et de grande taille. Le système de calcul symbolico-numérique proposé exploite systématiquement cet avantage en construisant, à partir des équations à résoudre une fonction **aTimesx**, et une liste ordonnée **b** de quantités numériques, exploités ensuite par la méthode **linearEquationRoot** d'un système de calcul numérique existant et répondant à la spécification retenue. Les intérêts de cette résolution itérative et *« matrix-free »* du système d'équations linéaires sont connus : le résultat obtenu est précis, et le coût du stockage en mémoire est limité par comparaison avec une méthode de résolution directe. En contre partie surgit la question de la convergence de la méthode itérative, liée aux caractéristiques spectrales de la matrice incidente et au choix de l'estimation de la solution.

La signature des méthodes de résolution des systèmes d'équations non linéaires diffère entre un système de calcul numérique et un système de calcul formel. La signature de la méthode **nonLinearEquationRoot** de la classe **ComputerAlgebraSystem** se compose de la liste des équations **equations**, transcrites sans transposition préalable, et de celle des symboles **variables**. Une valeur

estimée **initialGuess** d'une solution du système d'équations non linéaires peut également être saisie. La méthode **nonLinearEquationRoot** de la classe **NumericalCalculationSystem** résout un système d'équations non linéaires à partir de la donnée de deux fonctions, la fonction **residualEvaluation** calculant les résidus associés aux équations pour chaque valeur de **variables**, et la dérivée de cette fonction **jacobianEvaluation**, ainsi qu'une valeur estimée **initialGuess** de la solution. Cette interface est proposée par la plupart des bibliothèques mathématiques qui requièrent une fonction résidu, si possible complétée par une fonction calculant la matrice Jacobienne de cette fonction résidu.

Selon l'approche exposée dans la partie 2.2.1.2 le système de calcul symbolico-numérique proposé construit la fonction résidu et sa fonction dérivée. Celles-ci sont ensuite transmises à un code de résolution numérique existant, modélisé ici par la méthode **nonLinearEquationRoot** de la classe **NumericalCalculationSystem**. Dans de tels systèmes de calcul numérique la méthode de résolution est le plus souvent l'algorithme de Newton-Raphson. Chaque pas de Newton y est calculé par résolution d'un système linéaire dont la matrice incidente est la matrice Jacobienne de la fonction résidu. Or l'un des paragraphes précédents rappelle l'intérêt de résoudre un système linéaire à l'aide d'un système de calcul numérique où une méthode **linearEquationRoot** propose une résolution itérative et *« matrix-free »*. Il semble donc plus judicieux et cohérent de disposer d'une méthode **nonLinearEquationRoot** de résolution numérique de systèmes d'équations non linéaires ayant la signature suivante :

```
nonLinearEquationRoot   (
                  residualEvaluation : FunctionExpression,
                  residualDirectionalDerivativeEvaluation :
FunctionExpression,
                  initialGuess : Expression
                  ) : NonLinearEquationRootExpression
```

et non la signature proposée par la méthode **nonLinearEquationRoot** de la classe **NumericalCalculationSystem** :

```
nonLinearEquationRoot   (
                  residualEvaluation : FunctionExpression,
                  jacobianEvaluation : FunctionExpression,
                  initialGuess : Expression
                  ) : NonLinearEquationRootExpression
```

La méthode **nonLinearEquationRoot** utilisant non plus la matrice Jacobienne de la fonction résidu, mais plutôt la dérivée directionnelle de cette fonction résidu, est déclarée et définie dans la classe **SymbolicNumericalCalculationSystem**. Elle surcharge la méthode **nonLinearEquationRoot** déclarée dans la classe **ComputerAlgebraSystem**, et également définie dans la classe

SymbolicNumericalCalculationSystem. Outre la spécificité de son interface par rapport aux services proposés par les systèmes de calcul numérique existants, la méthode prend également en charge les systèmes d'équations sur et sous déterminés et les inéquations selon les principes exposés en 2.2.1.2. Dès lors, la résolution de systèmes d'équations non linéaires dans la classe **SymbolicNumericalCalculationSystem** exploite les services d'un système de calcul numérique existant, non pas lors de la résolution du problème initial, mais uniquement lors de la résolution des systèmes linéaires associés à chaque itération de Newton.

La méthode de résolution de systèmes d'équations algébro-différentielles **differentialAlgebraicEquationRoot** de la classe **NumericalCalculationSystem** exige la donnée de la fonction résidu **residualEvaluation**, de la fonction dérivée de celle-ci **jacobianEvaluation**, et de valeurs initiales cohérentes portant sur les variables dépendantes **dependentVariableStartingValues** et leurs dérivées **dependentVariableDerivativeStartingValues**. L'horizon d'intégration ainsi que certaines valeurs particulières de la variable indépendante composent la liste ordonnée **independentVariableValues**. La méthode **differentialAlgebraicEquationRoot** de la classe **ComputerAlgebraSystem** résout un ensemble d'équations algébro-différentielles **equations**. Cette liste inclut les conditions initiales exprimées sous la forme d'équations. La liste **dependentVariables** des symboles considérés comme des variables dépendantes définit les fonctions inconnues. Le symbole **independentVariable** repère la variable d'intégration. La liste ordonnée de valeurs numériques **independentVariableValues** définit l'horizon d'intégration, ainsi que certaines valeurs particulières de la variable indépendante pour lesquelles on souhaite connaître la valeur des variables dépendantes.

Selon le principe exposé en 2.2.1.3, le système de calcul symbolico-numérique proposé détermine à partir des équations fournies un ensemble de conditions initiales cohérentes. Celles-ci sont dissociées des équations algébro-différentielles proprement dites afin de construire la fonction résidu du système, puis la matrice Jacobienne de cette fonction résidu par rapport aux variables dépendantes et à leurs dérivées par rapport à la variable dépendante. Le système d'équations algébro-différentielles, ainsi transposé sous la forme de deux fonctions et d'une liste de valeurs initiales, est transmis à la méthode **differentialAlgebraicEquationRoot** d'un système de calcul numérique existant pour résolution. Afin de simuler des systèmes raides, la réalisation de cette méthode code un algorithme de type prédicteur-correcteur. Les phases de prédiction et de correction consistent toutes deux à résoudre un système d'équations non linéaires par appel de la méthode **nonLinearEquationRoot** de la classe **NumericalCalculationSystem**. Dans le cas de systèmes algébro-différentiels de grande taille et creux, il semble alors judicieux de remplacer les appels à la méthode **nonLinearEquationRoot** de la classe **NumericalCalculationSystem**, par des appels à la méthode **nonLinearEquationRoot**, définie dans la classe **SymbolicNumericalCalculationSystem**. La

méthode **differentialAlgebraicEquationRoot** de la classe **SymbolicNumericalCalculationSystem** peut alors accepter comme argument la dérivée directionnelle de la fonction résidu **residualDirectionalDerivativeEvaluation**, plutôt que la matrice Jacobienne de la fonction résidu. Le stockage d'une quelconque matrice creuse est ainsi évité tout au long de la résolution du système d'équations algébro-différentielles. Bien que séduisante, cette approche se heurte à une considération pratique : il est impossible de changer dans un intégrateur algébro-différentiel existant l'interface de la méthode de résolution de systèmes d'équations non linéaires sans modifier considérablement le code de la méthode **differentialAlgebraicEquationRoot**. Le travail présenté choisit dès lors d'utiliser des systèmes de calcul numérique existants pour résoudre les systèmes d'équations algébro-différentiels, l'apport original réside dans la construction automatique de la fonction résidu, dans le calcul formel de sa dérivée, dans le calcul de conditions initiales cohérentes et dans la réutilisation d'un code d'intégration numérique.

La méthode **minimum** de la classe **NumericalCalculationSystem**, recherchant le minimum local ou global d'un critère, exige la donnée d'une fonction **criterionEvaluation** calculant, pour une valeur quelconque des paramètres, la valeur du critère et la valeur de chacun des résidus correspondant aux contraintes mises sous une forme canonique (expressions mathématiques positives ou nulles). La fourniture de la fonction dérivée **criterionDerivativeEvaluation** de la fonction **criterionEvaluation** permet la mise en œuvre d'une méthode de résolution du premier ordre, voire du second ordre par approximation du Hessien. La liste ordonnée **initialGuess** estime la valeur des paramètres réalisant le minimum du critère. Dans le cas d'une optimisation sous contraintes, les paramètres formels **numberOfConstraints** et **numberOfEqualityConstraints** précisent respectivement le nombre total de contraintes et le nombre de contraintes de type égalité. Dans le cas d'un système de calcul formel, modélisé par la classe **ComputerAlgebraSystem**, la signature de la méthode **minimum** se compose de l'expression mathématique du critère **criterion**, de celle des contraintes de type égalité ou inégalité regroupées dans une liste **constraints**, de la liste ordonnée **variables** des symboles par rapport auxquels le minimum est recherché, et d'une estimation numérique **initialGuess** de la valeur de ces symboles pour laquelle le minimum recherché est atteint.

Selon le principe exposé en 2.2.1.4, le système de calcul symbolico-numérique proposé construit, à partir de l'expression mathématique du critère et des contraintes, les fonctions nécessaires à l'appel de la méthode **minimum** de la classe **NumericalCalculationSystem**. L'apport original de la méthode **minimum** de la classe **SymbolicNumericalCalculationSystem** réside dans la construction automatique des fonctions résidus associées aux contraintes, dans le calcul formel des dérivées du critère et des contraintes, et dans la réutilisation d'un code d'optimisation numérique existant.

Le système de calcul symbolico-numérique spécifié jusqu'à présent adopte l'interface des systèmes de calcul formel, entre autres en ce qui concerne la résolution de systèmes d'équations continues. Comme dans tout système de calcul formel, l'utilisateur formule son problème dans une syntaxe très proche de celle qu'il adopte dans un texte mathématique. Comme dans tout système de calcul numérique la solution proposée est une approximation numérique d'une solution exacte. En cela ce système de calcul symbolico-numérique n'est en rien original : la partie 1.1 rappelle que les systèmes de calcul formel, outre la résolution exacte de certains problèmes, proposent une résolution numérique approchée de la plupart des familles de systèmes d'équations continues. Comme cela a déjà été mis en avant dans le cadre général de l'évaluation symbolico-numérique d'expressions mathématiques, l'originalité de ce système de calcul réside dans sa coopération fine avec un système de calcul numérique existant lors de l'évaluation d'expressions mathématiques ou lors de la résolution d'équations continues. Cette coopération entre le système de calcul formel proposé et un système de calcul numérique existant se matérialise sur le diagramme de classes de la Figure 53 par l'association entre les classes **SymbolicNumericalCalculationSystem** et **NumericalCalculationSystem**. La partie 2.1.2.2 ayant limité à un le nombre de systèmes de calcul numérique susceptibles de coopérer avec le système de calcul formel proposé, cette association unidirectionnelle a pour multiplicité un. Le système de calcul numérique utilisé est transmis en argument du constructeur de la classe **SymbolicNumericalCalculationSystem**. Il est à noter que les expressions transmises au système de calcul numérique existant sont créées par un système de calcul formel, et sont donc évaluées par défaut suivant la sémantique d'un système de calcul formel décrite en 2.1.1.2. Dans certains cas, une évaluation de ces expressions par le système de calcul numérique exécutant l'algorithme de résolution est possible et souhaitable pour des raisons de performance. La méthode **clone**, déclarée dans la classe **Expression**, répond à ce besoin de performance en créant une expression identique à l'expression créée par le système de calcul formel, mais dont la sémantique est celle du système de résolution. Cependant, comme on le verra par la suite avec la notion de fonction implicite, conserver aux expressions créées par le système de calcul formel leur sémantique est parfois indispensable, et performant.

La partie 2.1.2 a montré que le modèle proposé pour une coopération entre calcul formel et calcul numérique s'établit autour de l'évaluation des expressions mathématiques créés par les différents systèmes de calcul coopérants. La partie 2.2.2 vient d'intégrer la résolution de problèmes continus dans ce modèle, en la considérant comme l'évaluation d'expressions mathématiques particulières. Des rôles prédéfinis ont été attribués à chacun des participants à la coopération -systèmes de calcul formel et systèmes de calcul numérique-, dans le but d'atteindre les objectifs de fiabilité du résultat et de performance lors du calcul. Ici, le modèle de

coopération, *« de type complémentaire »*, a été exploité dans un contexte où l'affectation des tâches de calcul à chacun des participants est statique. Toutefois, ce modèle de coopération demeure adapté à l'affectation dynamique des tâches grâce au mécanisme général de transfert de l'évaluation d'une expression, créée par un système de calcul, à un autre système de calcul.

2.2.3. Scénario de résolution d'un système d'équations non linéaires

Cette partie illustre la coopération entre le système de calcul formel proposé et un système de calcul numérique particulier, la bibliothèque mathématique IMSL, pour résoudre le système d'équations non linéaires $\forall(u,v) \in \Re^2; (u = 8 \cdot \cos(v)) \wedge (u + v = 6)$.

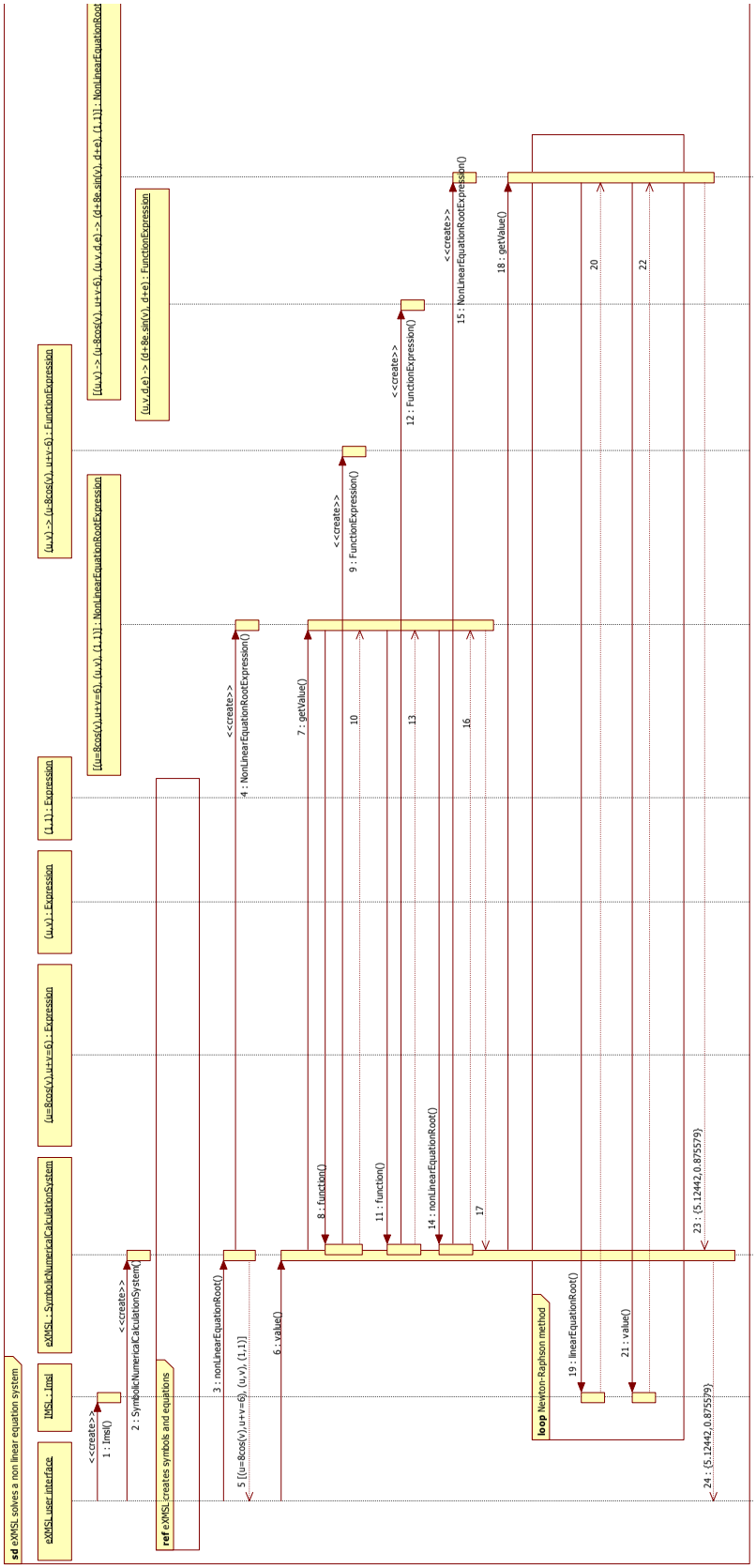

Figure 55 – Résolution symbolico-numérique du système $\forall(u,v)\in\Re^2;(u=8\cdot\cos(v))\wedge(u+v=6)$.

Le diagramme de séquence de la Figure 55 résume les principales étapes du processus de résolution du système d'équations non linéaires choisi. Le scénario commence par la création des deux systèmes de calcul employés lors de la résolution : un objet modélisant les services offerts par la bibliothèque mathématique IMSL et une instance du système de calcul symbolico-

numérique proposé, désignée par le terme eXMSL (eXtended Mathematical Software Library). Les stimuli de création des symboles **u**, **v**, **d** et **e** utilisés lors de ce processus ont été omis. De la même manière, les appels aux méthodes de l'objet « eXMSL » pour l'accès aux différentes expressions mathématiques constituant le système d'équations sont supposées décrites dans un autre diagramme « *eXMSL creates symbols and equations* » selon le principe général d'accès aux expressions détaillé précédemment. L'appel 3 à l'une des deux versions de la méthode **nonLinearEquationRoot** de l'objet « eXMSL » déclenche la construction par le système de calcul formel d'un objet représentant la résolution du système d'équations non linéaires. Cet objet, du type **NonLinearEquationRootExpression**, est composée de trois sous-expressions décrivant complètement le problème à résoudre : la liste des équations $(u = 8 \cdot \cos(v), u + v = 6)$, la liste des variables (u, v), et une estimation de la racine arbitrairement choisie égale à $(1,1)$. A partir des équations et des variables deux fonctions sont créées par le système de calcul formel : la fonction résidu $(u,v) \mapsto (u - 8 \cdot \cos(v), u + v - 6)$ et la fonction $(u,v,d,e) \mapsto (d + 8 \cdot e \cdot \sin(v), d + e)$ calculant, en un point (u,v) et selon une direction (d,e), la dérivée directionnelle de la fonction résidu. Des processus composés uniquement d'étapes de transformations formelles, non détaillées dans le diagramme de séquence, précèdent chacune des constructions de fonctions par les appels 9 et 12. La soustraction des membres droits des équations aux membres gauches des équations, puis la simplification algébrique de la liste d'expressions obtenue, précèdent la construction effective de la fonction résidu en 9. La différentiation des résidus, puis le remplacement des fonctions différentielles par les termes de la direction (d,e), et la simplification algébrique de la liste d'expressions obtenue précèdent la construction effective de la fonction dérivée directionnelle en 12. Les fonctions ainsi construites sont les arguments de l'appel 14 à la méthode **nonLinearEquationRoot** de l'objet « eXMSL ». Ce stimulus concerne la version de la méthode dont la signature est la suivante :

```
nonLinearEquationRoot   (
                  residualEvaluation : FunctionExpression,
                  residualDirectionalDerivativeEvaluation :
FunctionExpression,
                  initialGuess : Expression
                  ) : NonLinearEquationRootExpression
```

Cet appel déclenche la création d'une nouvelle expression de type **NonLinearEquationRootExpression**. Contrairement à l'appel 4 d'un constructeur de la classe **NonLinearEquationRootExpression** prenant comme arguments les équations et les variables, l'appel 15 du second constructeur de la classe transmet la fonction résidu et sa fonction dérivée directionnelle. La sémantique de la méthode **getValue** appelée en 18 s'en trouve changée :

l'algorithme itératif de Newton-Raphson est exécuté. La construction des différents systèmes linéaires et leur résolution sont confiés au système de calcul numérique IMSL via les stimuli 19 et 21. Une solution numérique approchée du problème $(5.12442, 0.875579)$ est produite et retournée successivement en 23 et en 24.

Dans le scénario décrit dans la Figure 55 la fonction résidu et sa dérivée directionnelle sont créées par eXMSL. A partir de ces expressions l'appel 18 à la méthode **getValue** de la classe **NonLinearEquationRootExpression** construit, à chaque itération de Newton de rang k, deux expressions mathématiques : la fonction dérivée directionnelle du résidu au point (u_k, v_k), définie par $(d,e) \mapsto (d + 8 \cdot e \cdot \sin(v_k), d + e)$, et l'opposé de l'évaluation de la fonction résidu au point (u_k, v_k), soit $(u_k - 8 \cdot \cos(v_k), u_k + v_k - 6)$. La construction de ces expressions, qui précède chaque séquence d'appels 19-20 peut, au choix, être demandée au système de calcul formel eXMSL, ou au système de calcul numérique IMSL. Dans ce cas précis, la construction par le système de calcul numérique des expressions intervenant dans la résolution des systèmes linéaires, puis leurs évaluations dans le contexte sémantique de ce système sont possibles et doivent être privilégiées. Cela est particulièrement vrai dans le cas de la fonction mathématique $(d,e) \mapsto (d + 8 \cdot e \cdot \sin(v_k), d + e)$. Chaque itération de Newton de rang k, résolue par un algorithme itératif du type *« matrix-free »*, impose plusieurs évaluations de l'expression $(d + 8 \cdot e \cdot \sin(v_k), d + e)$ pour différentes valeurs numériques de (d,e). Evaluer cette expression par des appels uniques à des routines informatiques de calcul numérique s'avère évidemment plus performant qu'appliquer de façon itérative des règles de transformations formelles à des expressions jusqu'à obtenir un point fixe.

Les avantages liés à l'usage du calcul formel apparaissent clairement sur cet exemple.

Le problème est exprimé dans une syntaxe proche d'un texte mathématique usuel. La spécification du problème n'est pas polluée par la spécification de la solution.

Le système de calcul prend en charge la construction de la fonction résidu et la construction de la fonction dérivée de la fonction résidu. L'algorithme de résolution bénéficie donc automatiquement des expressions requises par la méthode numérique mise en œuvre, et une étape préalable de dérivation manuelle, ou de synthèse de code par différentiation automatique, est évitée.

Un des avantages liés à l'usage du calcul numérique est manifeste sur l'exemple traité. La réutilisation d'un code mettant en œuvre une méthode usuelle de calcul numérique exploite des

savoir-faire algorithmique et mathématique existants. Un autre avantage souvent associé au calcul numérique, la performance en temps de calcul, n'apparaît pas dans cette étude élémentaire.

2.3. Simulation et optimisation symbolico-numérique de modèles implicites

Lors de la modélisation d'un système mécanique, thermodynamique ou multi-physique, les concepteurs de simulateurs continus établissent des équations où interviennent des symboles. Lorsqu'il s'agit de modéliser un système en régime permanent, ces symboles sont des scalaires, ou des fonctions de l'espace d'étude. Lorsque la dynamique du système est étudiée, ces symboles représentent des fonctions du temps et éventuellement de l'espace d'étude. Ces équations peuvent inclure des bilans matière ou énergie, des équations de transfert, etc. Outre ces équations, caractéristiques de l'état ou du comportement du système, des définitions fournissent la valeur numérique de certains symboles du modèle par évaluation numérique d'expressions mathématiques explicites comportant d'autres symboles calculés à l'aide des équations comportementales. Simuler un système continu consiste à affecter une valeur numérique à certains des symboles présents dans le modèle, puis à déterminer la valeur numérique des autres symboles par résolution d'équations algébriques, différentielles ou intégrales, ou par la minimisation d'un critère sous contraintes non linéaires. La partie 2.3 propose de considérer la simulation continue sous l'angle de l'évaluation d'une fonction implicite. Dès lors, l'optimisation du modèle, ou l'analyse de sensibilité sur ce dernier, requiert le calcul de dérivées de fonctions implicites. L'application d'un théorème des fonctions implicites permet une évaluation de ces dérivées a priori plus fiable que celle obtenue par des méthodes aux différences finies. Un gain de performance de l'approche proposée par rapport aux méthodes de différence finies n'est pas systématique. Il dépend de la nature du problème traité.

La partie 2.3.1 traite de l'évaluation des fonctions implicites définies par un système d'équations et d'inéquations non linéaires, et du calcul de leurs dérivées. La partie 2.3.2 applique un principe similaire pour la prise en compte des fonctions implicites définies par un système d'équations algébro-différentiel. La partie 2.3.3 s'intéresse aux fonctions implicites définies par la minimisation d'un critère sous contraintes non linéaires. La partie 2.3.4 complète le modèle de système de calcul symbolico-numérique par l'ajout de classes et de méthodes dédiées au traitement des fonctions implicites.

2.3.1. Fonction implicite définie par un système d'équations et d'inéquations non linéaires

L'état ou le comportement du système considéré dans cette partie est modélisé par un ensemble d'équations non linéaires. La spécification d'un problème de simulation particulier comprend alors deux étapes :

1. la désignation d'un certain nombre de symboles présents dans le modèle comme étant des entrées du système, et la désignation de tous les autres symboles comme étant des sorties du système.
2. l'affectation d'une valeur numérique à chacun des symboles d'entrée.

En appelant x le vecteur des variables d'entrée du système, et y le vecteur des variables de sortie du système, les équations du système peuvent être exprimées sous la forme fonctionnelle $F(x,y)=0$, où la fonction F est désignée comme étant la fonction modèle. Il s'agit, étant donné un vecteur x fourni en argument, de retourner le vecteur y satisfaisant l'équation du modèle. Cela définit une nouvelle fonction f que nous appelons la fonction module[17]. Dans de rares cas le modèle est explicite : le vecteur y peut directement être évalué à partir du vecteur x. Dans la plupart des cas l'équation du modèle est implicite : le vecteur y ne peut pas directement être évalué à partir du vecteur x, sa détermination requiert une procédure de résolution de l'équation. Près de la solution (x^*, y^*) de cette équation, il est possible de définir une fonction $f_{(x^*,y^*)}$ telle que, dans un voisinage $V_{(x^*,y^*)}$ de (x^*, y^*), toute solution (x,y) satisfasse $y = f_{(x^*,y^*)}(x)$. Plus précisément, en appelant i la taille du vecteur des variables d'entrée du système, et o la taille du vecteur des variables de sortie :

$$\forall (x^*, y^*) \in \Re^i \times \Re^o ; (F(x^*, y^*) = 0) \Rightarrow$$
$$(\exists V(x^*, y^*); \exists f_{(x^*,y^*)}; \forall (x,y) \in V(x^*, y^*); F(x,y) = 0 \Rightarrow y = f_{(x^*,y^*)}(x))$$

Équation 8 - Définition locale de la fonction module.

Cela signifie que localement une fonction module $f_{(x^*,y^*)}$ fournit y à partir de x. L'expression analytique de cette fonction reste bien évidemment inconnue, ce qui constitue un frein à sa prise en compte dans les outils de simulation. L'approche présentée ici, déjà évoquée dans (Joulia, Alloula, & Belaud 1999), puis mise en application dans (Alloula et al. 2006) et

[17] Dans un simulateur de procédés modulaire séquentiel cette fonction s'identifie au code d'un module, par exemple une opération unitaire représentant un élément physique du procédé.

(Alloula, Belaud, & Le Lann 2007), montre pourtant que la manipulation de cette fonction module tant en simulation qu'en optimisation est d'un grand intérêt.

Cette manipulation de la fonction module passe par la manipulation sous-jacente de la fonction modèle. La fonction modèle étant définie de $\Re^i \times \Re^o$ dans $\Re^o$, sa dérivée calculée en un point (x, y) de $\Re^i \times \Re^o$, $F'(x, y)$, peut être identifiée à une matrice réelle $(o \times (i + o))$:

$$F'(x,y) = \begin{pmatrix} \frac{\partial F_1}{\partial x_1}(x,y) & \cdots & \frac{\partial F_1}{\partial x_i}(x,y) & \frac{\partial F_1}{\partial y_1}(x,y) & \cdots & \frac{\partial F_1}{\partial y_o}(x,y) \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ \frac{\partial F_o}{\partial x_1}(x,y) & \cdots & \frac{\partial F_o}{\partial x_i}(x,y) & \frac{\partial F_o}{\partial y_1}(x,y) & \cdots & \frac{\partial F_o}{\partial y_o}(x,y) \end{pmatrix} = \left[\partial_1 F(x,y) \quad \partial_2 F(x,y)\right]$$

Équation 9 – Matrice Jacobienne de la fonction modèle.

La matrice $F'(x, y)$ peut être vue comme une matrice par blocs, constituée de deux sous-matrices $\partial_1 F(x, y)$ et $\partial_2 F(x, y)$.

La simulation de modèles consiste à évaluer la fonction module, c'est-à-dire à résoudre un système d'équations non linéaires. La plupart du temps cette résolution est itérative et basée sur la méthode de Newton-Raphson. A chaque itération, outre la valeur des résidus $F(x, y_k)$ et une évaluation de leur norme $\|F(x, y_k)\|$, cette résolution utilise une sous-matrice $\partial_2 F(x, y_k)$ de la matrice Jacobienne du modèle.

L'analyse de sensibilité, ou l'optimisation du modèle par rapport à certains paramètres d'entrée s'appuie sur la dérivée de la fonction module. La valeur $f'(x)$ de la dérivée de la fonction module en un point x peut être identifiée à une matrice réelle $(o \times i)$:

$$f'(x) = \begin{pmatrix} \frac{\partial f_1}{\partial x_1}(x) & \frac{\partial f_1}{\partial x_2}(x) & \ldots & \frac{\partial f_1}{\partial x_i}(x) \\ \ldots & \ldots & \ldots & \ldots \\ \ldots & \ldots & \ldots & \ldots \\ \frac{\partial f_o}{\partial x_1}(x) & \frac{\partial f_o}{\partial x_2}(x) & \ldots & \frac{\partial f_o}{\partial x_i}(x) \end{pmatrix}$$

Équation 10 – Matrice de sensibilité.

Si cette matrice est la plupart du temps estimée par perturbations numériques, ou calculée par différentiation automatique du code de simulation du modèle, nous proposons son calcul à partir d'un des théorèmes des fonctions implicites.

La fonction G, définie de $\Re^i$ dans $\Re^o$, par : $G(x) = F(x, f(x))$ est la fonction nulle. Sa dérivée $G'(x)$ en tout point x est la fonction nulle de $L(\Re^i, \Re^o)$, ensemble des applications linéaires de $\Re^i$ dans $\Re^o$, identifiable à la matrice nulle $(o \times i)$. La dérivation de la fonction G en un point x conduit à l'équation fonctionnelle :

$$\partial_1 F(x, f(x)) + \partial_2 F(x, f(x)).f'(x) = 0$$

Équation 11 – Calcul de la matrice de sensibilité $f'(x)$ au point x.

Il apparaît alors que l'obtention de la dérivée de la fonction module, ou matrice de sensibilité du module, consiste à résoudre un ensemble de i systèmes d'équations linéaires de taille $(o \times o)$ dont les inconnues sont les colonnes de la matrice de sensibilité $f'(x)$ et dont la matrice incidente est toujours $\partial_2 F(x, y)$. Lors de la résolution par une méthode directe, cette matrice pourra être factorisée une seule fois, et chaque colonne de $f'(x)$ calculée par une élimination de Gauss.

Le calcul formel de la matrice $\partial_2 F(x, y)$, puis son évaluation numérique précise en différents points (x, y) doivent contribuer à la qualité numérique de la solution $(x, f(x))$ obtenue. Le calcul formel de la matrice Jacobienne de la fonction modèle, puis son évaluation numérique précise au point solution $(x, f(x))$ doivent contribuer à la qualité numérique de la matrice de sensibilité $f'(x)$. Du fait de la méthode de calcul retenue pour celle-ci, cette qualité numérique dépend également de la fiabilité du code choisi pour la résolution numérique de systèmes linéaires.

Le calcul de chaque colonne de la matrice $f'(x)$ par une méthode aux différences finies requiert la résolution d'un système de o équations non linéaires à o inconnues, associé à la perturbation d'une des composantes du vecteur x. Le calcul de la matrice $f'(x)$ par une méthode aux différences finies requiert au total la résolution de i systèmes de o équations non linéaires à o inconnues. La méthode alternative proposée pour le calcul de la matrice de sensibilité d'une fonction implicite définie par un système d'équations non linéaires est donc toujours plus performante qu'une méthode aux différences finies.

Etant donné un vecteur ϕ quelconque de $\Re^i$, l'équation fonctionnelle précédente peut être réécrite sous la forme :

$$\partial_1 F(x, f(x)) \cdot \phi + \partial_2 F(x, f(x)).f'(x) \cdot \phi = 0$$

Équation 12 – Calcul de la sensibilité directionnelle $f'(x) \cdot \phi$ au point x selon la direction ϕ.

Cette forme est adaptée au calcul de la sensibilité directionnelle $f'(x) \cdot \phi$ au point x et selon une direction ϕ. Le calcul de cette sensibilité consiste à résoudre un système linéaire par une méthode directe ou itérative. Si l'on dispose d'une méthode d'évaluation de la dérivée directionnelle $\partial_2 F(x, y) \cdot \varphi$ en tout point (x, y) et selon toute direction φ, une méthode de résolution itérative et *« matrix-free »* pourra être employée.

2.3.2. Fonction implicite définie par un système d'équations algébro-différentielles

La partie 2.2.1.3 donne la forme la plus générale d'un système d'équations algébro-différentielles $\begin{cases} F(t, y(t), y'(t)) = 0 \\ y(t_0) = y_0 \end{cases}$, où $F : U \subset I \times \Re^n \times \Re^n \to \Re^n$, U est un ouvert, I est un intervalle de $\Re$ et $(t_0, y_0) \in I \times \Re^n$. Les variables dépendantes y sont vues comme des fonctions de l'unique variable indépendante t. Certains symboles, autres que t, t_0 et y apparaissent dans cette formulation, notamment le symbole y_0. Tous ces symboles ont nécessairement une valeur numérique définie par avance de sorte que le système d'équations algébro-différentielles peut être intégré. Il est pourtant intéressant de considérer ces symboles, non plus comme des valeurs numériques, mais plutôt comme les m paramètres p d'une famille de systèmes d'équations algébro-différentielles $\begin{cases} F(p, t, y(p,t), D_2 y(p,t)) = 0 \\ y(p, t_0) = g(p) \end{cases}$, où $F : U \subset \Re^m \times I \times \Re^n \times \Re^n \to \Re^n$, U est un ouvert, I est un intervalle de $\Re$, $t_0 \in I$, $g : V \subset \Re^m \to \Re^n$ et V est un ouvert. Dès lors, étant donnés une valeur particulière des paramètres p^*, et une valeur particulière de la variable indépendante t^*, la valeur de la matrice de sensibilité des variables dépendantes par rapport aux paramètres exprimée au point (p^*, t^*), soit $D_1 y(p^*, t^*)$, se calcule assez aisément par application du théorème des fonctions implicites.

La dérivation du système paramétrique d'équations algébro-différentielles par rapport aux paramètres conduit au système paramétré d'équations algébro-différentielles suivant :

$$\begin{cases} D_1F(p,t,y(p,t),D_2y(p,t))+ \\ D_3F(p,t,y(p,t),D_2y(p,t))\cdot D_1y(p,t)+ \\ D_4F(p,t,y(p,t),D_2y(p,t))\cdot D_1D_2y(p,t)=0 \\ \qquad D_1y(p,t_0)=g'(p) \end{cases}$$

En désignant par s les fonctions dérivées partielles des variables dépendantes par rapport aux paramètres, soit D_1y, et en tenant compte de l'identité des fonctions D_1D_2y et D_2D_1y, le système précédent devient :

$$\begin{cases} D_1F(p,t,y(p,t),D_2y(p,t))+ \\ D_3F(p,t,y(p,t),D_2y(p,t))\cdot s(p,t)+ \\ D_4F(p,t,y(p,t),D_2y(p,t))\cdot D_2s(p,t)=0 \\ \qquad s(p,t_0)=g'(p) \end{cases}$$

La matrice de sensibilité $s(p^*,t^*)=D_1y(p^*,t^*)$ est alors obtenue par intégration numérique sur l'horizon $\lfloor t_0,t^* \rfloor$ du système d'équations algébro-différentielles suivant, correspondant à la valeur particulière p^* des paramètres :

$$\begin{cases} F(p^*,t,y(p^*,t),D_2y(p^*,t))=0 \\ D_1F(p^*,t,y(p^*,t),D_2y(p^*,t))+ \\ D_3F(p^*,t,y(p^*,t),D_2y(p^*,t))\cdot s(p^*,t)+ \\ D_4F(p^*,t,y(p^*,t),D_2y(p^*,t))\cdot D_2s(p^*,t)=0 \\ y(p^*,t_0)=g(p^*) \\ \qquad s(p^*,t_0)=g'(p^*) \end{cases}$$

Équation 13 – Calcul de la matrice de sensibilité $s(p^*,t^*)$ au point (p^*,t^*).

Dans ce système d'équations algébro-différentielles, les fonctions inconnues sont les variables dépendantes y et les sensibilités s.

L'évaluation précise et performante de la matrice de sensibilité $s(p^*,t^*)$ est un enjeu important car elle conditionne la qualité de toute optimisation d'un système régi par un modèle algébro-différentiel. Le calcul formel des fonctions dérivées D_1f, D_3f, D_4f et g', puis l'évaluation numérique précise du résultat de l'application de ces fonctions en différents points, doit contribuer à l'obtention précise de la matrice de sensibilité $s(p^*,t^*)$. Dans ce cadre, l'utilisation du système de calcul symbolico-numérique proposé dispense de toute stratégie de perturbation des paramètres associée à une méthode de différences finies. Elle devrait donc

limiter l'erreur de calcul des sensibilités aux seules erreurs liées à la méthode d'intégration numérique.

La comparaison de performance avec une approximation purement numérique de la matrice de sensibilité est en revanche défavorable à la méthode proposée. L'application du théorème des fonctions implicites conduit à la résolution d'un seul système de $n+m\cdot n$ équations algébro-différentielles, et à l'obtention des $n\cdot m$ valeurs constituant la matrice de sensibilité. L'utilisation d'une méthode de différences finies permet d'obtenir, en perturbant la valeur d'un seul des paramètres, une seule colonne de la matrice de sensibilité par l'intégration du système paramétrique initial de n équations algébro-différentielles. L'obtention de la matrice de sensibilité complète requiert alors l'intégration de m systèmes d'équations algébro-différentielles de n équations chacun. Le coût de l'intégration numérique de systèmes d'équations algébro-différentielles étant plus que linéaire, résoudre m systèmes de n équations algébro-différentielles est moins coûteux que résoudre un seul système de $m\times n$ équations algébro-différentielles, donc a fortiori moins coûteux que résoudre un seul système de $n+m\cdot n$ équations algébro-différentielles.

$$\begin{cases} F(p^*,t,y(p^*,t),D_2y(p^*,t))=0 \\ D_1F(p^*,t,y(p^*,t),D_2y(p^*,t))\cdot d+ \\ D_3F(p^*,t,y(p^*,t),D_2y(p^*,t))\cdot\sigma(p^*,t,d)+ \\ D_4F(p^*,t,y(p^*,t),D_2y(p^*,t))\cdot D_2\sigma(p^*,t,d)=0 \\ y(p^*,t_0)=g(p^*) \\ \qquad\qquad \sigma(p^*,t_0,d)=g'(p^*)\cdot d \end{cases}$$

Équation 14 – Calcul de la sensibilité directionnelle $\sigma(p^*,t^*,d)$ au point (p^*,t^*) suivant la direction d.

Une alternative intéressante au calcul d'une matrice de sensibilité complète $s(p^*,t^*)$, par l'intégration de $n+m\cdot n$ équations algébro-différentielles, est l'intégration d'un système algébro-différentiel aboutissant à une valeur d'une sensibilité directionnelle $s(p^*,t^*).d$. En désignant par σ la fonction sensibilité directionnelle, définie par $\sigma:\begin{array}{l}\Re^m\times I\times\Re^m\to\Re^n\\(p,t,d)\mapsto s(p,t)\cdot d\end{array}$, étant donné $d\in\Re^m$, la sensibilité directionnelle $\sigma(p^*,t^*,d)$ au point (p^*,t^*) suivant la direction d est obtenue par l'intégration du système algébro-différentiel Équation 14. L'obtention de la valeur d'une sensibilité directionnelle par application d'un théorème des fonctions implicites requiert donc une intégration d'un système de $2n$ équations algébro-différentielles, contre une intégration d'un système de n équations algébro-différentielles pour le calcul par un schéma aux

différences finies. Le gain de précision attendu de la méthode symbolico-numérique peut alors justifier un surcoût en temps de calcul désormais raisonnable. L'usage des sensibilités directionnelles n'a de sens cependant que si l'algorithme d'optimisation manipule directement les sensibilités directionnelles et non des produits de matrices de sensibilité et de directions.

2.3.3. Fonction implicite définie par la minimisation d'un critère sous contraintes non linéaires

La démarche développée dans la partie précédente s'applique aux fonctions implicites définies par la minimisation d'un critère différentiable. Le problème de la recherche d'un minimum d'un critère J sous contraintes s'écrit $\min_u J(u)$ sous $\theta(u) \in -C$, où $J : \Re^n \to \Re$, $\theta : \Re^n \to \Re^m$, avec $m < n$, et C un cône convexe de $\Re^m$ qui sera généralement pris égal à $\{0\} \subset \Re^m$ (contraintes égalité) ou égal à $\left(\Re^+\right)^m$ (contraintes inégalité). Certains symboles peuvent apparaître dans l'expression analytique du critère J, mais ces derniers ont nécessairement une valeur numérique définie par avance, de sorte que toute valeur numérique des variables d'optimisation u définit une valeur particulière du critère. Il est pourtant intéressant de considérer ces symboles, non plus comme des valeurs numériques, mais plutôt comme les q paramètres p d'une famille de problèmes de minimisation $\min_u J(p,u)$ sous $\theta(p,u) \in -C$, où $J : \Re^q \times \Re^n \to \Re$, $\theta : \Re^q \times \Re^n \to \Re^m$, avec $m < n$, et C un cône convexe de $\Re^m$. Sous réserve de l'existence d'un ouvert $V \subset \Re^q$ tel que l'existence et l'unicité du minimum $\min_u J(p,u)$ est garantie pour toute valeur p dans V, il est possible de définir une fonction implicite f par : $f : \begin{array}{c} V \subset \Re^q \to \Re^n \\ p \mapsto \min_u J(p,u) \text{ sous } \theta(p,u) \in -C \end{array}$. La fonction implicite f étant définie par un calcul de minimum et non par une relation algébrique, aucune hypothèse de régularité faite sur J et θ ne permet de déduire de résultats de régularité sur la fonction f. Néanmoins, f est désormais supposée continûment différentiable. Dès lors, étant donnée une valeur particulière des paramètres p^*, la valeur de la matrice de sensibilité des variables dépendantes par rapport aux paramètres exprimée au point p^*, soit $f'(p^*)$, se calcule assez aisément par application d'un théorème de fonctions implicites.

2.3.3.1. Minimisation sans contraintes

Dans un premier temps l'hypothèse $C = \Re^m$, signifiant que le critère n'est pas contraint, simplifie l'exposé. Dans ce cas, la condition nécessaire du premier ordre permettant de caractériser la solution du problème d'optimisation associé à la valeur $p \in V$ des paramètres s'écrit $D_2 J(p, f(p)) = 0$. Elle stipule que le gradient du critère J s'annule en un extrémum local. La condition nécessaire est valable pour tout $p \in V$. Sa dérivation par rapport aux paramètres conduit au système d'équations non linéaires suivant :

$$\forall p \in V; D_1 D_2 J(p, f(p)) + D_2 D_2 J(p, f(p)) \cdot f'(p) = 0$$

Etant donnée une valeur particulière des paramètres p^*, l'équation suivante relie la sensibilité $f'(p^*)$ du minimum global du critère J par rapport aux paramètres exprimée au point p^* et certaines dérivées partielles de J :

$$D_1 D_2 J(p^*, f(p^*)) + D_2 D_2 J(p^*, f(p^*)) \cdot f'(p^*) = 0$$

Équation 15 – Calcul de la matrice de sensibilité $f'(p^*)$ au point p^*.

Etant donnée une valeur particulière des paramètres p^*, l'obtention de la matrice de sensibilité $f'(p^*)$ consiste à résoudre un ensemble de q systèmes d'équations linéaires dont les inconnues sont les colonnes de la matrice de sensibilité $f'(p^*)$ et dont la matrice incidente est toujours la matrice $D_2 D_2 J(p^*, f(p^*))$, sous-matrice de la matrice Hessienne $\mathrm{H}J(p^*, f(p^*))$ du critère évaluée au point $(p^*, f(p^*))$. Lors de la résolution par une méthode directe, cette matrice pourra être factorisée une seule fois, et chaque colonne de $f'(p^*)$ calculée par une élimination de Gauss.

Etant donné un vecteur ϕ quelconque de $\Re^q$, l'équation fonctionnelle précédente peut être réécrite sous la forme :

$$D_1 D_2 J(p^*, f(p^*)) \cdot \phi + D_2 D_2 J(p^*, f(p^*)) \cdot f'(p^*) \cdot \phi = 0$$

Équation 16 – Calcul de la sensibilité directionnelle $f'(p^*) \cdot \phi$ au point p^* selon la direction ϕ.

Cette forme est adaptée au calcul de la sensibilité directionnelle $f'(p^*) \cdot \phi$ au point p^* et selon une direction ϕ. Le calcul de cette sensibilité consiste à résoudre un système linéaire par une méthode directe ou itérative. Si l'on dispose d'une méthode d'évaluation de la dérivée

directionnelle $D_2D_2J(x,y)\cdot\varphi$ en tout point (x,y) et selon toute direction φ, une méthode de résolution itérative et *« matrix-free »* pourra être employée.

2.3.3.2. Minimisation sous contraintes égalité

Le théorème des multiplicateurs de Lagrange propose des conditions nécessaires du premier ordre dans le cas de contraintes égalité. Dans le cadre de la formulation paramétrique adoptée, la condition nécessaire du premier ordre permettant de caractériser la solution $f(p)$ du problème d'optimisation associé à la valeur $p\in V$ des paramètres s'écrit :

$$D_2J(p,f(p))+\sum_{i=1}^{m}\lambda_i(p)\cdot D_2\theta_i(p,f(p))=0$$

où $\lambda(p)$ est le vecteur des multiplicateurs de Lagrange, fonction du vecteur p des paramètres. Plus précisément, étant donné $p\in V$, le système de $n+m$ équations non linéaires constitué par les conditions nécessaires du premier ordre et les contraintes égalité peut permettre de calculer simultanément un extrémum $\bar{f}(p)$ du critère et le vecteur des multiplicateurs de Lagrange $\lambda(p)$ associé.

$$\begin{cases} D_2J(p,\bar{f}(p))+\sum_{i=1}^{m}\lambda_i(p)\cdot D_2\theta_i(p,\bar{f}(p))=0 \\ \theta(p,\bar{f}(p))=0 \end{cases}$$

Équation 17 – Calcul d'un extrémum $\bar{f}(p)$ du critère et du vecteur des multiplicateurs de Lagrange $\lambda(p)$ associé.

Si cet extrémum $\bar{f}(p)$ est un minimum du critère, alors $f(p)=\bar{f}(p)$ et le problème d'optimisation est résolu. Pour tout $p\in V$, le système d'équations suivant est donc vérifié :

$$\begin{cases} D_2J(p,f(p))+\sum_{i=1}^{m}\lambda_i(p)\cdot D_2\theta_i(p,f(p))=0 \\ \theta(p,f(p))=0 \end{cases}$$

En supposant désormais connue la valeur de la fonction implicite f en tout point $p\in V$, la dérivation des équations précédentes par rapport aux paramètres p fournit un système de $q\times n+q\times m$ équations linéaires. Les inconnues de ce système linéaire sont les $q\times n$ termes de la matrice Jacobienne $f'(p)$ et les $q\times m$ termes de la matrice Jacobienne $\lambda'(p)$.

$$\begin{cases} D_1 D_2 J(p, f(p)) + D_2 D_2 J(p, f(p)) \cdot f'(p) + \\ \lambda'(p)^T \cdot D_2 \theta\big(p, f(p)\big) + \lambda(p)^T \cdot \big(D_1 D_2 \theta\big(p, f(p)\big) + D_2 D_2 \theta\big(p, f(p)\big) \cdot f'(p)\big) = 0 \\ D_1 \theta(p, f(p)) + D_2 \theta(p, f(p)) \cdot f'(p) = 0 \end{cases}$$

Équation 18 – Calcul de la sensibilité $f'(p)$ du minimum du critère sous contraintes égalité.

La dérivation partielle formelle du critère et des contraintes, puis la résolution numérique du système linéaire caractéristique, produit une valeur numérique approchée de la sensibilité $f'(p)$ du minimum du critère sous contraintes égalité. La fiabilité de la valeur ainsi obtenue est directement liée à la qualité de la résolution du système linéaire précédent.

La performance du calcul de $f'(p)$ par résolution d'un système linéaire de dimension $q \times (m+n)$ est à évaluer par comparaison avec l'application d'une méthode de différences finies. Cette dernière nécessite la résolution de q problèmes d'optimisation comportant n variables et m contraintes égalité, correspondant chacun à une perturbation d'une des composantes du vecteur p. Sous l'hypothèse initiale $m < n$, le nombre q de paramètres et le nombre n de variables d'optimisation conditionnent en partie le choix de la méthode de calcul de la sensibilité du critère. Mais comparer a priori le temps de calcul nécessaire à la résolution de q problèmes d'optimisation de taille $m+n$ et le temps de calcul nécessaire à la résolution d'un système linéaire de dimension $q \times (m+n)$ est difficile.

2.3.3.3. Minimisation sous contraintes inégalité

Les conditions nécessaires du première ordre, dites de Karush, Kuhn et Tucker ne sont pas directement exploitables pour le calcul de la sensibilité $f'(p)$ du minimum du critère sous contraintes inégalité. En effet, la dérivation de chacun des deux termes des contraintes inégalité $\theta\big(p, f(p)\big) \leq 0$ par rapport aux paramètres p ne permet pas d'obtenir une relation entre ceux-ci. Pour ce calcul uniquement, et selon le principe déjà appliqué dans la partie 2.2.1.2 à la résolution de systèmes d'équations et d'inéquations non linéaires, la formulation du problème de minimisation peut être transformée par l'ajout de variables d'écart élevées au carré dans les contraintes inégalité. Le problème de minimisation sous contraintes inégalité devient alors un problème de minimisation sous contraintes égalité et la démarche précédente s'applique.

2.3.4. Modèle structurel et comportemental des fonctions implicites et de leurs dérivées

Tous les systèmes de calcul formel actuels proposent la notion de fonction dont l'expression analytique est fournie explicitement. Sur le modèle des fonctions du lambda-calcul la fonction n'est pas nécessairement nommée, et les paramètres formels s'ils sont nommés peuvent être substitués par d'autres symboles. Maple propose la notion de procédure, similaire à celle des routines de tout langage procédural. La possibilité d'une notation voisine de la notation mathématique usuelle démarque cependant le système de calcul formel d'un langage procédural généraliste : x -> x^2 désigne par exemple la fonction élevant son unique argument au carré. Sur le même principe, Axiom dispose de fonctions anonymes et adopte une notation similaire à celle de Maple. Mathematica propose la notion de *« pure function »* dans laquelle les paramètres formels n'ont même pas à être énumérés dans la déclaration, mais apparaissent dans la définition sous la désignation conventionnelle **#1**, **#2**, … Dans le modèle structurel de système de calcul construit dans ce chapitre les objets fonctions, présents aussi bien dans des systèmes de calcul formel que dans des systèmes de calcul numérique, sont des instances de la classe **FunctionExpression**.

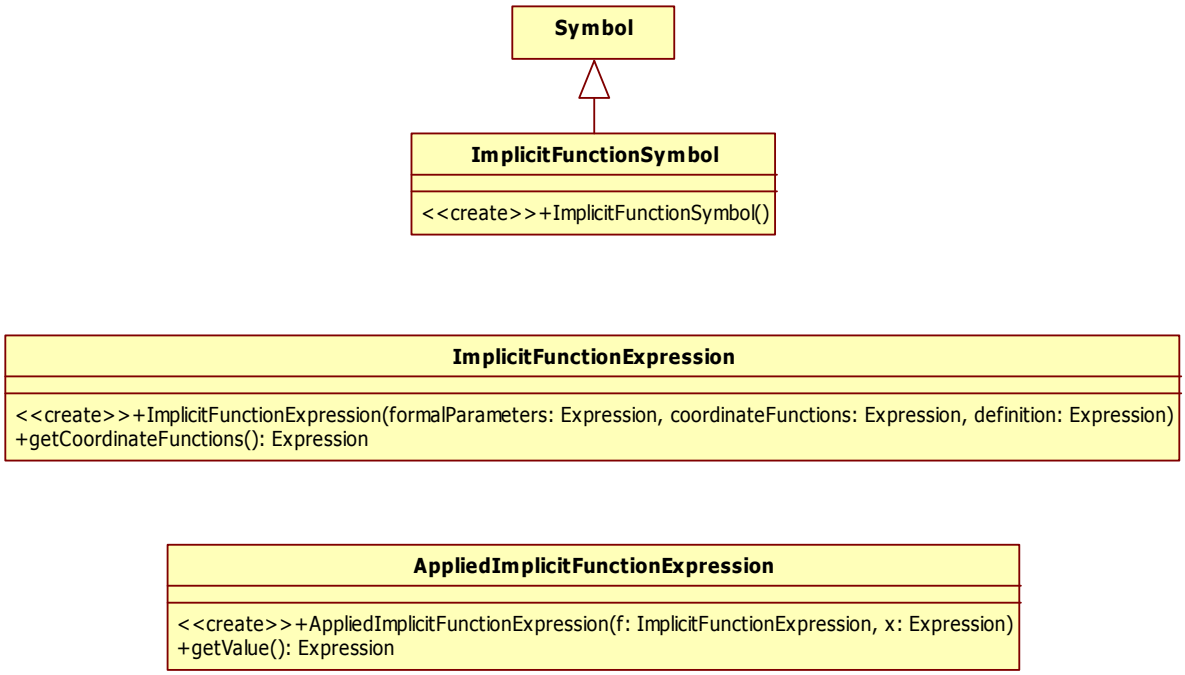

Figure 56 – Modèle structurel pour la définition de fonctions implicites dans un système de calcul.

Une des originalités principales du système de calcul symbolico-numérique proposé par rapport aux systèmes de calcul existants est l'introduction de la notion de fonction implicite. Dans le modèle structurel de ce système les fonctions implicites sont des instances de la classe **ImplicitFunctionExpression**. Le diagramme de classes de la Figure 56 précise l'interface de la classe **ImplicitFunctionExpression** ainsi que les classes supplémentaires requises **ImplicitFunctionSymbol** et

AppliedImplicitFunctionExpression. L'unique constructeur de la classe **ImplicitFunctionExpression** fait apparaître la structure des expressions fonctions implicites. Celles-ci sont composées de trois sous-expressions : une liste des paramètres formels, une liste des fonctions coordonnées et la définition de la fonction implicite permettant de calculer la valeur des fonctions coordonnées pour une valeur numérique particulière des paramètres formels. La définition de la fonction implicite dans le système de calcul exposé est :

- soit une expression du type **NonLinearEquationRootExpression**, si les fonctions coordonnées sont calculées par résolution d'un système d'équations et d'inéquations non linéaires ;
- soit une expression du type **DifferentialAlgebraicEquationRootExpression**, si les fonctions coordonnées sont calculées par résolution d'un système d'équations algébro-différentielles ;
- soit une expression du type **MinimumExpression**, si les fonctions coordonnées sont calculées par minimisation d'un critère sous contraintes non linéaires, de types égalité ou inégalité.

La classe **AppliedImplicitFunctionExpression** modélise l'application d'une fonction implicite en un point. Contrairement à la méthode **getValue** de la classe **AppliedFunctionExpression**, la méthode **getValue** de la classe **AppliedImplicitFunctionExpression** est concrète. En effet la sémantique de celle-ci est complètement définie. L'algorithme de la méthode comprend trois étapes successives :

1. remplacer dans la définition de la fonction implicite chaque occurrence d'un paramètre formel par la valeur numérique de l'argument correspondant ;
2. résoudre le problème ainsi obtenu ;
3. remplacer chaque symbole de la liste des fonctions coordonnées par la valeur numérique correspondante calculée lors de la résolution. La liste des valeurs numériques produite constitue le résultat de la méthode **getValue**.

Les méthodes **getValue** des classes **PartialDerivativeExpression**, **JacobianMatrixExpression**, **HessianMatrixExpression** et **DirectionalDerivativeExpression** sont enrichies afin de mettre en œuvre le calcul des sensibilités des fonctions implicites tel qu'il est exposé dans les parties 2.3.1, 2.3.2 et 2.3.3.

Il est à noter que les objets de type **AppliedImplicitFunctionExpression** sont des expressions. A ce titre ils peuvent apparaître dans toute expression mathématique, notamment dans la définition d'une autre fonction implicite. Dès lors, l'intérêt de représenter les fonctions implicites, et leurs évaluations en certains points, devient évident. Contrairement à la démarche habituelle des systèmes de modélisation et de simulation orientés équations qui, suite à la modélisation, combinent les équations des différents sous-systèmes pour constituer le système d'équations complet à simuler, la modélisation à l'aide de fonctions implicites n'impose pas une

mise à plat des équations avant leur résolution. La simulation peut, au choix, être simultanée ou hiérarchique et parallèle. Dans ce dernier cas, la simulation d'un système peut être vue comme l'évaluation de la fonction implicite qui lui est associée, cette évaluation conduisant à l'évaluation d'autres fonctions implicites et de leurs dérivées : les fonctions implicites modélisant chaque sous-système.

La manipulation de sous-systèmes en tant que fonctions implicites est un cadre naturel pour simuler des systèmes c'est-à-dire évaluer des fonctions, pour optimiser des systèmes c'est-à-dire dériver des fonctions, pour combiner des systèmes c'est-à-dire composer ou définir des fonctions. Par ailleurs la définition de chaque fonction implicite à partir d'autres fonctions implicites structure fortement la distribution éventuelle des calculs sur plusieurs processeurs. Lors de la simulation d'un modèle M_1 une fonction implicite f_1 est évaluée en un point x_1. Cette évaluation consiste à résoudre le système d'équations associé ou le problème d'optimisation associé. Si les équations, les inéquations ou le critère comportent des évaluations de fonctions implicites $f_{1,1}$, $f_{1,2}$, …, $f_{1,n}$ alors l'évaluation de $f_1(x_1)$ par un algorithme itératif utilisant des dérivées d'ordre un requiert, à chaque itération k, les évaluations des quantités numériques $f_{1,1}\left(x_{1,1}^{(k)}\right)$, $f_{1,2}\left(x_{1,2}^{(k)}\right)$, …, $f_{1,n}\left(x_{1,n}^{(k)}\right)$, $f'_{1,1}\left(x_{1,1}^{(k)}\right)$, $f'_{1,2}\left(x_{1,2}^{(k)}\right)$, …, $f'_{1,n}\left(x_{1,n}^{(k)}\right)$. Les quantités numériques $f_{1,1}\left(x_{1,1}^{(k)}\right)$, $f_{1,2}\left(x_{1,2}^{(k)}\right)$, …, et $f_{1,n}\left(x_{1,n}^{(k)}\right)$ sont indépendantes, et peuvent donc être calculées en parallèle. Les quantités numériques $f'_{1,1}\left(x_{1,1}^{(k)}\right)$, $f'_{1,2}\left(x_{1,2}^{(k)}\right)$, …, $f'_{1,n}\left(x_{1,n}^{(k)}\right)$ sont également indépendantes et peuvent donc être calculées en parallèle. Toutefois, le calcul de sensibilité d'une fonction implicite en un point exploite la valeur de la fonction en ce point. Le calcul de $f_{1,1}\left(x_{1,1}^{(k)}\right)$ précède donc celui de $f'_{1,1}\left(x_{1,1}^{(k)}\right)$, le calcul de $f_{1,2}\left(x_{1,2}^{(k)}\right)$ celui de $f'_{1,2}\left(x_{1,2}^{(k)}\right)$, etc.

La modélisation de systèmes complets fait souvent apparaître des cycles entre les sous-systèmes. Du point vue fonctionnel, cela se traduit par des fonctions implicites dont les arguments sont des évaluations d'autres fonctions implicites. Dans ce cas, deux stratégies extrêmes sont possibles :

1. Estimer les valeurs de tous les arguments de chaque fonction implicite, évaluer toutes les fonctions implicites en ces points, considérer l'écart entre les valeurs évaluées de certains symboles et leurs valeurs estimées. Si cet écart est trop important, modifier la valeur estimée de ces symboles à partir des sensibilités des fonctions implicites en ces points, et itérer. Cette stratégie est adoptée dans les simulateurs modulaires séquentiels de procédés continus, où le cycle est rompu en au moins un point, et les caractéristiques du « courant coupé » sont approchées de manière itérative. Dans le domaine du génie des procédés, cette démarche est notamment abordée dans (Joulia, Koehret, & Enjalbert 1985) ;
2. Remplacer les fonctions implicites associées à chaque sous-système présent dans le cycle par une fonction implicite unique construite par concaténation des équations caractéristiques de chaque sous-système. Cette stratégie est adoptée dans les simulateurs de procédés (continus)

orientés équations, tels que gPROMS (Oh & Pantelides 1996;Process Systems Enterprise Ltd. 2000) ou Dymola (Dempsey 2006), qui fusionnent l'ensemble des équations de tous les sous-systèmes constitutifs du système étudié pour procéder ensuite à une résolution globale du modèle obtenu.

La modélisation d'un système à l'aide de fonctions implicites doit contribuer à la fiabilité des simulations ou des optimisations de ce système. D'une part, chaque évaluation de fonction implicite, c'est-à-dire chaque résolution, peut tirer le meilleur parti d'une initialisation appropriée au modèle. D'autre part, la taille réduite des modèles, par rapport à une résolution globale, limite les mauvais conditionnements notamment la raideur des systèmes d'équations algébro-différentielles à traiter.

Une conséquence moins évidente de l'usage des fonctions implicites est d'introduire une synchronisation naturelle de la simulation des sous-systèmes par les données. Le modèle d'exécution « data-flow », classique en calcul parallèle, surgit dès lors que l'évaluation numérique d'une fonction implicite en un point commence uniquement lorsque toutes les coordonnées de ce point ont été évaluées numériquement. Ce modèle d'exécution « data-flow », couplé au modèle de système de calcul symbolico-numérique proposé[18], pourrait constituer le cadre d'un environnement de simulation de systèmes dynamiques, continus ou hybrides.

[18] (Whiting & Pascoe 1994) montre le lien étroit entre les langages « data-flow » et les langages fonctionnels, langages à l'origine des systèmes de calcul formel.

2.4. Apports complémentaires au modèle de conception d'un système de calcul symbolico-numérique

La partie 2.4 s'intéresse à la réutilisation dans le système de calcul proposé, réutilisation vue comme une contribution pour atteindre les objectifs de fiabilité et de performance. La lecture de cette partie n'est en rien impérative pour la compréhension du manuscrit. Par conséquent, l'essentiel du texte est reporté en annexe.

La réutilisation des codes de calcul existants, abordée en Annexe A, vise à restituer précisément une connaissance métier, souvent accumulée et disséminée dans des routines multiples écrites dans des langages procéduraux compilés.

La réutilisation des expressions mathématiques créées par le système de calcul, traitée en Annexe B, vise un gain de performance en évitant de créer de multiples instances d'expressions identiques, et en exploitant le résultat d'évaluations précédentes.

Les techniques employées ici –interface avec des fonctions boîtes noires, partage de sous-expressions communes, et cache d'évaluation- ne sont pas nouvelles. L'originalité du travail présenté tient à la modélisation des entités intervenant dans ces techniques, puis à la fusion des différents modèles construits avec le modèle conceptuel de système de calcul symbolico-numérique construit au Chapitre 2.

2.5. Synthèse

Le Chapitre 2 établit un modèle de coopération de type complémentaire entre des systèmes de calcul formel et des systèmes de calcul numérique. Le système global vise deux objectifs : la fiabilité des résultats obtenus, et la performance. L'hypothèse simplificatrice adoptée, selon laquelle un système de calcul formel unique coopère avec un système de calcul numérique unique, ne réduit en rien la généralité du modèle présenté, mais sert la clarté de l'exposé.

Deux principes fondent le modèle de coopération développé :

1. les modèles, ainsi que les spécifications des problèmes à résoudre, sont identifiés à des expressions mathématiques, qui constituent dès lors les données communes aux différents systèmes de calcul ;

2. la sémantique d'évaluation d'une expression dépend du système de calcul responsable de la tâche d'évaluation. Ce principe est supporté par un mécanisme original de clonage d'une expression dans le système de calcul chargé de l'évaluer.

Dans le cadre de ces deux principes, le modèle de coopération applique une stratégie où :

1. à partir d'une expression représentant un modèle, le système de calcul formel produit automatiquement toutes les expressions mathématiques requises par la méthode numérique de résolution. A chaque demande d'un système de calcul numérique, il évalue numériquement, en un point, une des expressions construites au préalable ;

2. l'évaluation d'une expression représentant une spécification de problème conduit à l'exécution d'une tâche par un système de calcul numérique, c'est-à-dire à la mise en œuvre d'une méthode numérique particulière.

Chapitre 3. eXMSL, une mise en œuvre du modèle de coopération entre calcul formel et calcul numérique

Le modèle de conception d'un système de calcul symbolico-numérique, établi au Chapitre 2, donne lieu ici à un modèle d'implémentation, puis à des réalisations logicielles.

La partie 3.1 expose les transformations qui conduisent du modèle de conception à un modèle d'implémentation, puis celles qui permettent de générer, à partir du modèle d'implémentation, des modules dans le langage FORTRAN 90.

La partie 3.2 décrit deux réalisations logicielles issues de l'application des règles de transformation précitées, en s'intéressant en particulier aux services offerts à l'utilisateur.

3.1. Transformation du modèle de conception en modèle d'implémentation

Le modèle de conception du système de calcul symbolico-numérique construit dans le Chapitre 2, n'est pas directement exécutable sur une machine, qu'elle soit réelle ou virtuelle. La disponibilité d'un programme exécutable exige une transposition de ce modèle en un modèle d'implémentation, où un choix technologique a eu lieu pour chaque entité du modèle de conception. Les choix technologiques sont de plusieurs ordres, et dépendent en partie de la plate-forme logicielle et matérielle à partie de laquelle sera construite la solution. A titre d'exemple, si la plate-forme logicielle choisie pour l'implémentation est Java 2 Standard Edition, un choix d'implémentation est l'utilisation de la classe *HashTable* proposée par J2SE pour mettre en œuvre tout ensemble de données accessible par une clé, qui apparaît dans le modèle de conception.

La partie 3.1.1 énonce les principaux avantages et inconvénients de la plate-forme d'implémentation choisie. Les parties qui suivent présentent les principaux choix d'implémentation effectués lors de la transformation du modèle de conception d'un système de calcul symbolico-numérique en un prototype logiciel : eXMSL.

3.1.1. Avantages et inconvénients de la plate-forme FORTRAN 90

Le choix de la « plate-forme »[19] FORTRAN 90 pour le codage du modèle d'implémentation du système proposé présente plusieurs avantages. Un système de calcul crée diverses expressions intermédiaires, dont le nombre et la nature sont inconnus a priori. FORTRAN 90 permet d'allouer des structures de données en cours d'exécution pour ces expressions (allocation dynamique de la mémoire), et de les libérer dès lors que les expressions n'ont plus à être référencées. Le système de calcul symbolico-numérique proposé est aussi un système de calcul numérique : l'évaluation des fonctions mathématiques de base, telles que les fonctions trigonométriques, pour des arguments numériques doit avoir lieu dans le système lui-même, sans être sous-traitée à un système de calcul numérique tiers. FORTRAN 90 propose une palette de fonctions mathématiques usuelles intégrées au langage, et souvent optimisées pour les architectures matérielles cibles. Le système de calcul symbolico-numérique proposé veut tirer parti des environnements de simulation numérique existants en s'intégrant avec eux. Beaucoup

[19] Même si le terme « plate-forme » est utilisé dans ce cas, il apparaît abusif en comparaison de la richesse de plates-formes comme Java 2 Enterprise Edition ou Microsoft .NET.

de ces environnements, notamment les bibliothèques mathématiques numériques en libre accès, sont écrites en FORTRAN. Les développements de logiciels scientifiques actuels utilisent également les langages C et C++. Interfacer, au niveau du langage de programmation, ces outils avec du code écrit en FORTRAN 90 est un processus maîtrisé et proposé par les environnements de développement logiciel. Le système de calcul symbolico-numérique proposé vise la performance, tant du point de vue du temps d'exécution que du point de vue de la gestion des structures de données en mémoire. Les compilateurs FORTRAN 90 génèrent des codes compilés, optimisés sur ces deux plans :

1. outre l'accès à la vectorisation, les compilateurs actuels mettent à disposition du développeur un parallélisme SMP[20] via des directives Open-MP ;
2. comparée à des plates-formes orientées objets tels que Java, l'environnement d'exécution d'une application FORTRAN 90 est réduit.

A ces avantages il convient d'ajouter certaines facilités offertes par le langage qui ont un intérêt particulier dans le contexte du développement d'une application scientifique : les primitives de manipulation des tableaux possèdent une sémantique riche, et la possibilité de surcharge des opérateurs arithmétiques permet la manipulation d'expressions symbolico-numériques dans une syntaxe identique à celle utilisée lors de la transcription d'expressions uniquement numériques. Par ailleurs, l'intégration de codes scientifiques existants, souvent développés en FORTRAN 77, en FORTRAN 90 ou en C, est naturelle dans ce contexte.

Enfin, ma maîtrise et mon goût pour ce langage ont constitué un argument supplémentaire en faveur de ce choix.

Si les arguments en faveur de l'adoption du langage FORTRAN 90 sont importants, trois types d'inconvénients au moins sont liés à l'utilisation d'un langage procédural lors du développement logiciel concerné :

1. le langage FORTRAN 90 ne possède pas toutes les caractéristiques d'un langage orienté objet. Même si les notions de module et de visibilité assurent l'encapsulation des données, il ne fournit pas de mécanisme d'héritage, de polymorphisme et de liaison dynamique ;
2. l'absence de processus automatiques dans le cycle de vie du logiciel : pas de génération automatique de code FORTRAN 90 à partir de spécifications au format UML, pas d'outil de

[20] Le parallélisme à mémoire partagée (SMP pour « Shared Memory Parallelization ») repose sur l'exécution parallèle de plusieurs processus légers accédant à une mémoire globale. *« L'interface la plus répandue pour le contrôle d'une mémoire globale est Open-MP. Open-MP permet l'annotation d'un code séquentiel écrit dans un langage standard (Fortran, C ou C++) par des directives permettant de préciser le type de parallélisme et la stratégie d'ordonnancement associée. »*

rétro-ingénierie produisant une spécification au format UML à partir de codes sources FORTRAN 90 ;

3. la difficulté d'interfacer, au niveau du code de programmation, des langages à objets avec le langage FORTRAN 90. L'accès, indépendant du langage de programmation, aux services proposés par le système de calcul symbolico-numérique justifie le développement d'un composant logiciel, évoqué dans la partie 3.2.2.

3.1.2. Transformation des classes de conception en modules FORTRAN 90

3.1.2.1. Règles de transformation d'un modèle de conception en modèle d'implémentation UML

Afin de formaliser en partie la transformation d'un modèle de conception en modèle d'implémentation utilisant la plate-forme FORTRAN 90, les règles suivantes de transformation sont énoncées :

1. une hiérarchie de classes dans le modèle de conception donne naissance à un module unique dans le modèle d'implémentation[21] ;
2. un module du modèle d'implémentation regroupe tous les attributs et toutes les méthodes des différentes classes du modèle de conception qu'il représente. Le regroupement des attributs de toutes les classes d'une hiérarchie dans une classe unique a un inconvénient majeur : la création de tout objet à partir de cette classe unique alloue systématiquement de l'espace mémoire pour toutes les références éventuelles, donc la plupart du temps pour de nombreuses références qui resteront inutilisées. Seul un langage disposant du mécanisme d'héritage permet d'adapter le nombre de références allouées aux besoins des objets créés ;
3. les classes/modules du modèle d'implémentation sont toutes estampillées du stéréotype **<<FORTRAN module>>** ;
4. pour chaque module **M** du modèle d'implémentation, représentant une hiérarchie complète de classes du modèle de conception, est ajouté un module **MTypes** énumérant et distinguant les différents types dans la hiérarchie de conception. Le module **MTypes** définit autant de constantes entières ayant une valeur distincte qu'il existe de sous-classes dans la hiérarchie représentée. Ces constantes entières indépendantes de toute instance de **M** sont définies comme des attributs de classe publics ;
5. le statut de visibilité des membres d'une classe de conception est légèrement modifié dans les classes d'implémentation : un membre privé reste privé, un membre public reste public, un membre « protected » devient privé du fait de l'absence d'héritage dans la plate-forme cible ;
6. une relation de composition entre deux classes de conception implique la fusion de la classe composant dans la classe composite au niveau du modèle d'implémentation ;

[21] Ce choix peu élégant est presque dicté par l'absence de la notion d'héritage dans le langage FORTRAN 90. Des tentatives pour simuler ce mécanisme, caractéristique des langages à objets, ont eu lieu (Decyk, Norton, & Szymanski 1998). Leur complexité de mise en œuvre contrebalance l'avantage apporté par la structuration hiérarchique.

7. les cycles de dépendances entre classes sont rompus. Cette transformation est requise par les contraintes sémantiques du langage FORTRAN 90, qui interdisent l'importation mutuelle de deux modules par utilisation du mot-clé **USE**. Dès lors si deux classes du modèle de conception sont reliées entre elles de façon bidirectionnelle, soit par une association, soit par une agrégation, soit par une relation de dépendance stéréotypée **<<access>>**, **<<import>>** ou **<<instantiate>>**, le modèle d'implémentation se doit de modéliser différemment ces liens. Cette transformation d'une partie du modèle de conception vers une partie du modèle d'implémentation demeure à la charge de l'architecte logiciel car elle s'avère difficilement automatisable. En effet, outre l'information structurelle, les relations de dépendance supposent souvent une information sémantique dont il faut également tenir compte lors de la transformation.

3.1.2.2. Règles de transformation d'un modèle d'implémentation UML en modules FORTRAN 90

Afin de s'inscrire dans le contexte sémantique du langage cible, les conventions suivantes sont adoptées lors de la génération du code FORTRAN 90 à partir du modèle d'implémentation :

8. un module est généré pour chaque classe stéréotypée **<<FORTRAN module>>**. Le nom d'un module est le nom de sa classe d'origine, auquel est ajouté le caractère distinctif « _ » ;
9. dans chaque module un type dérivé est défini. Le nom du type dérivé est le nom de sa classe d'origine. La convention adoptée ci-dessus pour les noms des modules évite un conflit entre le nom d'un type dérivé et le nom du module où il est défini. Les champs d'un type dérivé sont les attributs de la classe d'origine. Le modificateur d'accès **PRIVATE** appliqué à l'ensemble des champs du type dérivé les rend inaccessibles directement en dehors du module ;
10. la manipulation des objets par référence impose de déclarer avec le mot-clé **POINTER** toutes les données de types dérivés dans le code généré ;
11. FORTRAN 90 ne permet pas la déclaration directe de tableaux de références vers des données de types dérivés. Dans ce but, un nouveau type dérivé doit être déclaré ; ce type dérivé encapsule un champ qui est une référence vers une instance du type dérivé défini en 9. Si la classe du modèle d'implémentation se nomme **Xyz**, le type dérivé est nommé **XyzArrayElement** ;
12. le mécanisme de surcharge de FORTRAN 90 autorise la définition de plusieurs versions d'une routine, se distinguant entre elles selon leur prototype. Mais ces différentes versions ne peuvent cependant pas être définies dans le module avec le même nom. La génération de code FORTRAN 90 doit renommer les différentes versions d'une méthode définie dans une classe stéréotypée **<<FORTRAN module>>** et déclarer un bloc interface déclarant le nom présent dans le modèle d'implémentation comme nom générique des différentes routines ;
13. les dépendances stéréotypées **<<import>>**, **<<access>>** ou **<<instantiate>>** se traduisent par une importation du module cible dans le module source ;

14. les attributs de classe[22] dont la valeur initiale est figée sont traduits en données constantes, déclarées avec le mot-clé **PARAMETER** ;
15. le paradigme objet est inversé lors de la génération du code FORTRAN 90 correspondant aux méthodes d'instance. Contrairement aux langages à objets où les méthodes sont appelées à partir d'un objet donné, les méthodes des classes stéréotypées **<<FORTRAN module>>** sont appelées par leur nom uniquement. La dépendance de ces méthodes vis-à-vis des objets, ou structures de données, s'exprime par la présence systématique d'un premier argument dont le type est celui défini dans la classe. L'appel d'une méthode d'instance à partir d'un objet, qui s'exprime sous la forme **objet.méthode(arguments)** dans un langage à objets, est traduit sous la forme **méthode(objet , arguments)** dans un langage procédural ;
16. les méthodes de classe ne subissent pas de transformation au-delà du changement de syntaxe ;
17. les constructeurs sont traités de manière un peu particulière : aucun paramètre formel n'est ajouté à la liste initiale, le nom des constructeurs est préfixé avec la chaîne de caractères « new » afin d'éviter un conflit de nom avec le type dérivé associé au module, enfin un suffixe différent ajouté aux noms des différents constructeurs évite tout conflit de noms entre ces derniers. Le nom générique des constructeurs est le nom de la classe de conception préfixé avec la chaîne de caractères « new » ;
18. le destructeur subit une transformation unique au-delà du changement de syntaxe : son nom **finalize** est suffixé avec le nom de la classe de conception afin d'éviter un conflit de nom avec le type dérivé associé au module.

3.1.2.3. Application au modèle de conception du système de calcul symbolico-numérique

En appliquant les règles de transformation énoncées dans la partie précédente, les différentes classes présentes dans les diagrammes de classes UML présentées auparavant sont traduites en plusieurs classes d'implémentation. La correspondance n'est pas bijective : aux nombreuses classes du modèle de conception ne correspondent que six classes d'implémentation dans le diagramme UML de la Figure 57 : **Expression**, **ExpressionTypes**, **HashTableEntry**, **HashTable**, **CalculationSystem** et **CalculationSystemTypes**.

[22] Les attributs et les méthodes de classes, dits statiques, indépendants de toute instance, apparaissent soulignés dans les diagrammes de classes UML présentés.

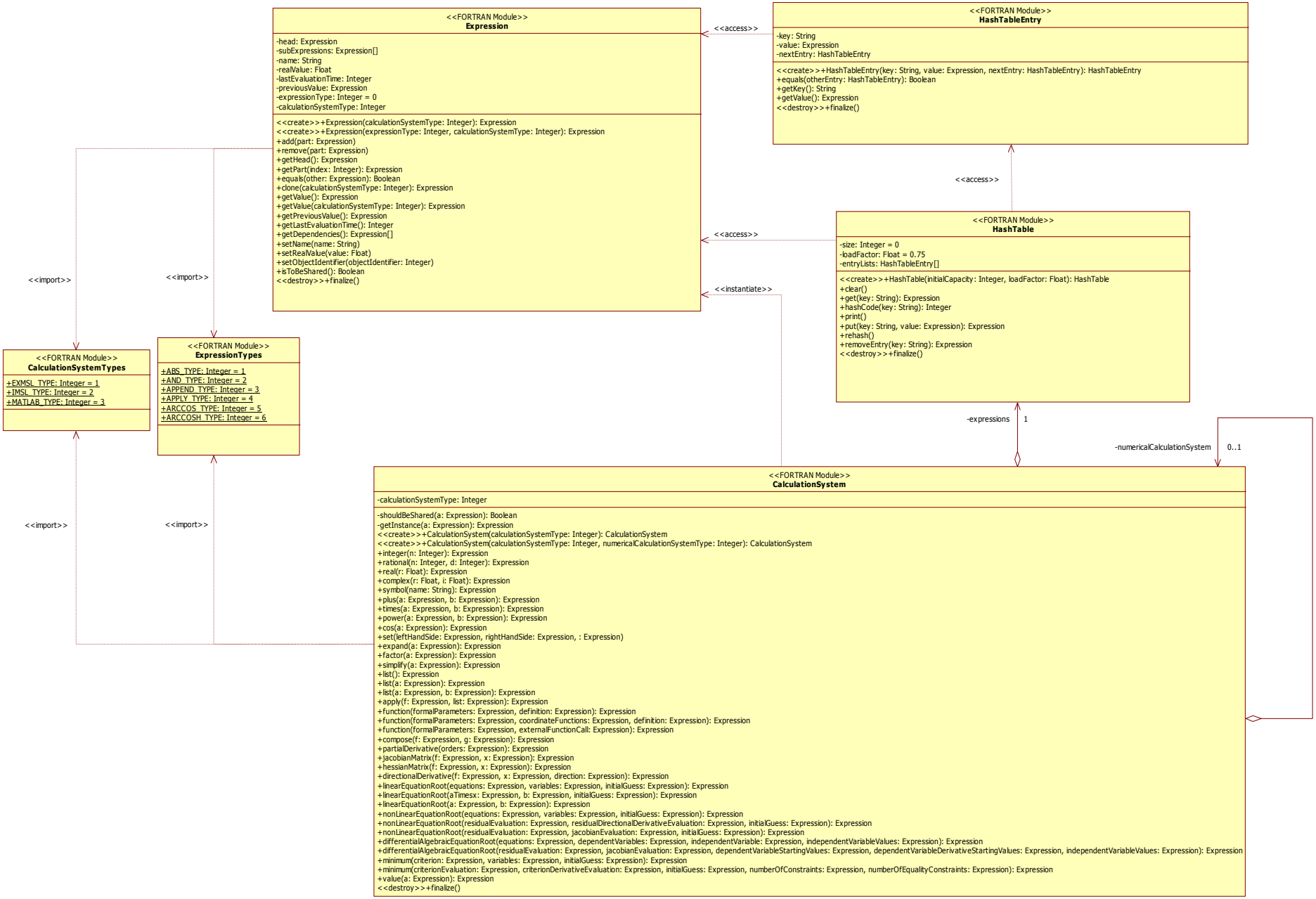

Figure 57 – Modèle d'implémentation du système de calcul symbolico-numérique proposé.

Par application de la règle 1 les classes **Expression**, **CompositeExpression**, **AtomicExpression**, **Symbol**, **Integer**, …, du modèle de conception sont transposées en une unique classe **Expression** dans le modèle d'implémentation. De manière similaire, les classes **CalculationSystem**, **ComputerAlgebraSystem**, **SymbolicNumericalCalculationSystem** et **NumericalCalculationSystem** du modèle de conception sont transposées en une unique classe **CalculationSystem** dans le modèle d'implémentation.

L'application de la règle 3 permet, dans le modèle d'implémentation, à la classe **ExpressionTypes** de garder une trace des différentes sous-classes de conception de la classe **Expression**. La liste des différents types d'expressions permet un traitement différencié dans les méthodes du module **Expression** selon la valeur du champ **expressionType**. Il en est de même pour la classe **CalculationSystemTypes**.

Dans le modèle d'implémentation, la dépendance mutuelle des classes **Expression** et **CalculationSystem** devient impossible. La règle 7 nous invite à rompre le cycle de dépendances entre les deux classes de conception. En fait, la dépendance de la classe **Expression** par rapport à la classe **CalculationSystem** est excessive : une expression a seulement à connaître le type du système de calcul qui l'a créée, et non le système de calcul particulier qui l'a créée. Dès lors, le rôle **evaluators** de l'association entre les classes **Expression** et **CalculationSystem** du modèle de conception est remplacé par un attribut **calculationSystemType**, dont la valeur précise le type du système de calcul ayant créé l'expression courante.

Outre les transformations systématiques introduites ci-dessus, des choix d'implémentation particuliers ont été faits indépendamment de la plate-forme cible.

Le référencement de certaines des expressions créées par un système de calcul prend la forme d'une table de hachage dont les clés sont la transcription, en langage MathML 2.0 de contenu, des expressions. Le module **HashTable** représente les tables de hachage dont les entrées sont des instances de la classe **HashTableEntry**. Le langage FORTRAN 90 ne proposant pas la généricité, les entrées insérées dans les tables de hachage sont typées a priori : dans notre cas précis les clés sont choisies comme étant des chaînes de caractères et les valeurs sont nécessairement des expressions.

3.1.3. Transposition de la méthode d'évaluation **value** du système de calcul

Comme cela a été détaillé dans la partie 2.1.2.2, l'algorithme général d'évaluation dans un système de calcul numérique diffère de celui d'un système de calcul formel. De plus, des

stratégies différentes de partage des sous-expressions communes peuvent par exemple différencier l'algorithme de deux systèmes de calcul formel. De ce fait, la fonction **value** du module **CalculationSystem_** exécute un code différent selon la valeur de l'attribut caché **calculationSystemType**. Une structure de contrôle **SELECT CASE** aiguille vers la traduction en FORTRAN 90 de la méthode **value** d'une des sous-classes de la classe de conception **CalculationSystem**.

3.1.4. Transposition des méthodes d'évaluation getValue des différents types d'expression

Sur le même principe que la fonction **value** du module **CalculationSystem_**, la fonction **getValue** du module **Expression_** exécute un code différent selon la valeur de l'attribut caché **expressionType**. Une structure de contrôle **SELECT CASE** aiguille vers la traduction en FORTRAN 90 de la méthode **getValue** d'une des sous-classes de la classe de conception **Expression**.

3.1.5. Mise en œuvre du mécanisme de partage des sous-expressions communes

Le modèle de système de calcul symbolico-numérique veut tirer le meilleur parti du mécanisme de partage des sous-expressions communes en référençant systématiquement les expressions qui doivent être uniques dans le système, et celles dont il est souhaitable de référencer un exemplaire afin d'exploiter au mieux le mécanisme de cache d'évaluation ultérieurement. La fonction **isToBeShared** du module **Expression** retourne la valeur vraie si et seulement si **(expressionType = SYMBOL_TYPE)**. La fonction **shoulBeShared** du module **CalculationSystem_** doit trouver le meilleur compromis lors d'une séquence de calcul particulière, sur une plate-forme logicielle et matérielle particulière, entre le stockage d'expressions dans la table de hachage **expressions** et la réévaluation éventuelle d'expressions non stockées. Ce compromis, qui définit l'algorithme précis de la fonction **shoulBeShared**, doit être trouvé pour chaque contexte d'étude particulier. En effet, il diffère non seulement selon les contraintes d'espace mémoire et de temps de calcul imposées mais aussi selon la séquence de calculs réalisés. L'étude précise de ce compromis sera l'objet de travaux ultérieurs.

3.2. Utilisation du système de calcul symbolico-numérique

La génération de code FORTRAN 90 à partir du modèle d'implémentation produit le système de calcul symbolico-numérique objet du travail présenté. Ce système de calcul coopératif est complété par des éléments logiciels qui le rendent accessible à partir d'environnements de simulation numérique :

- sous la forme d'une bibliothèque mathématique FORTRAN 90 décrite dans la partie 3.2.1 ;
- sous la forme d'un composant logiciel de type service Web dont la partie 3.2.2 précise l'architecture.

Nous exhibons trois rôles principaux pour l'utilisation de ces outils : « basic user », « model builder » et « model developer ». L'Annexe C formalise les interactions entre ces acteurs et la bibliothèque mathématique ou les composants logiciels, au travers d'un diagramme de cas d'utilisation[23].

3.2.1. Bibliothèque mathématique symbolico-numérique

3.2.1.1. Surcharge des opérateurs arithmétiques et des fonctions intrinsèques FORTRAN 90

Le modèle d'implémentation de la partie 3.1 est complété par un certain nombre de classes stéréotypées **<<FORTRAN module>>** qui facilitent l'usage de la bibliothèque eXMSL à partir de clients FORTRAN 90. Le module **OperatorOverloading_** propose une surcharge des opérateurs arithmétiques du langage FORTRAN 90 afin que ceux-ci acceptent des opérandes du type **Expression**. Sur le même principe, le module **IntrinsicFunctionOverloading_** propose une surcharge des fonctions intrinsèques du langage FORTRAN 90 afin que celles-ci acceptent des arguments du type dérivé **Expression**. Le module **BuiltInFunctionOverloading_** surcharge certaines fonctions publiées par le module **CalculationSystem_** afin que celles-ci puissent être appelées non seulement avec des arguments du type dérivé **Expression**, mais aussi avec des arguments de types primitifs du langage FORTRAN 90, comme des tableaux de nombres réels. Enfin le module **AssignmentOverloading_** interface la fonction **set** du module **CalculationSystem_** de sorte que le signe = puisse affecter une expression à une autre.

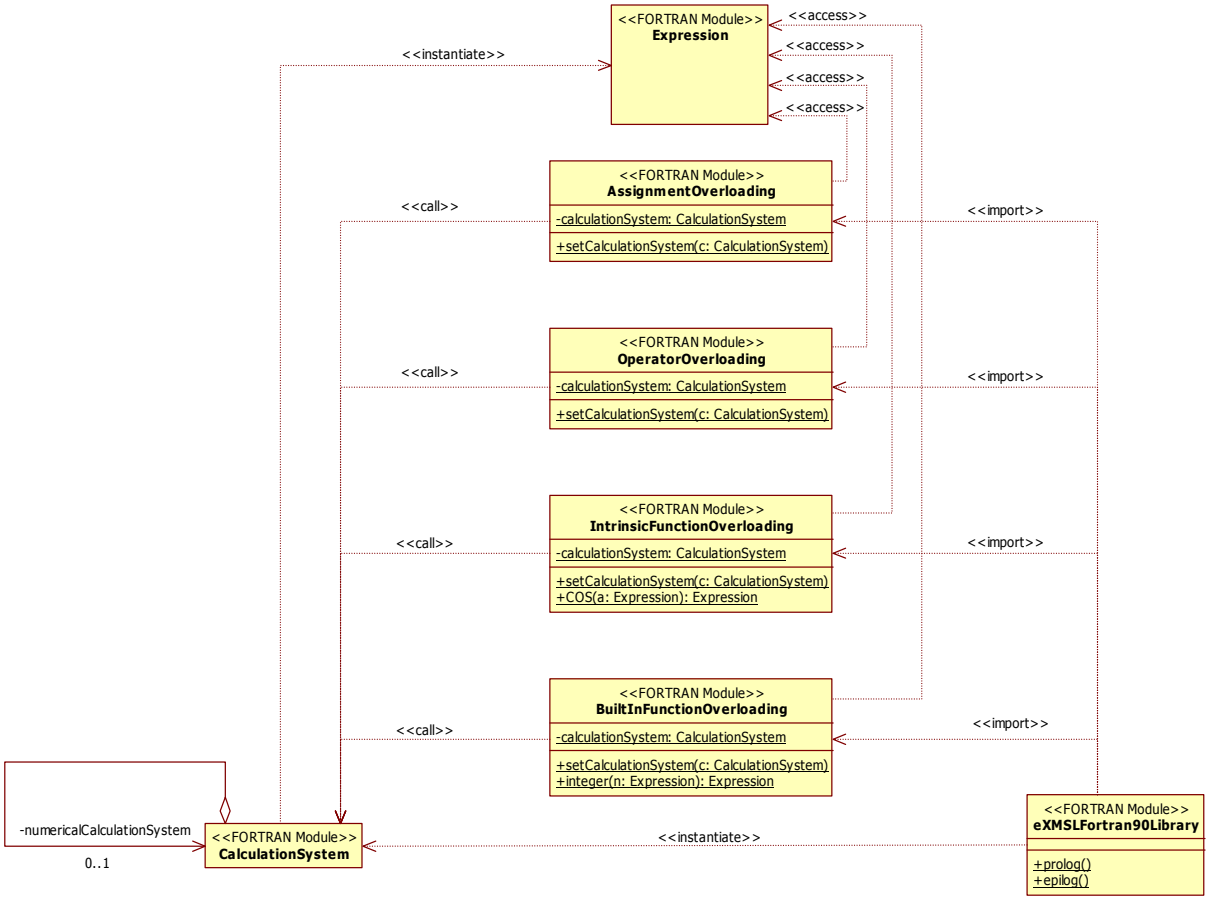

Figure 58 – Modèle d'implémentation de la bibliothèque mathématique symbolico-numérique eXMSL FORTRAN 90 Library : accès à partir d'un client FORTRAN 90.

Le diagramme de classes de la Figure 58 rend compte de l'usage du mécanisme de surcharge pour proposer à l'utilisateur de la bibliothèque eXMSL une syntaxe d'appel des fonctions et des opérateurs de calcul symbolico-numérique la plus proche possible de la syntaxe FORTRAN 90. Il est à noter que l'ensemble des fonctions publiées par la classe **CalculationSystem** se trouve surchargé au niveau de la classe **eXMSL**. Toutes les méthodes d'instance sont transformées en méthodes de classe de sorte que, par application de la règle de transformation 16, énoncée dans la partie 3.1.2.1, le prototype des fonctions FORTRAN 90 générées à partir du modèle d'implémentation comporte un nombre de paramètres formels identique à celui des opérateurs arithmétiques ou des fonctions intrinsèques du langage. Dès lors, ces opérateurs arithmétiques et ces fonctions intrinsèques sont appelés avec des arguments qui sont des expressions symbolico-numériques de la même manière que lorsqu'ils sont appelés avec des arguments qui sont des variables numériques.

[23] Au cours de stages de recherche, des élèves ingénieurs ont exploité le système eXMSL FORTRAN 90 Library en tant que « model developer ». Il est envisagé d'utiliser en enseignement eXMSL on the Web dans un rôle de « model builder ».

3.2.1.2. Appel à une bibliothèque mathématique numérique : IMSL

La bibliothèque mathématique IMSL est un des systèmes de calcul numérique choisi pour coopérer avec le système de calcul formel développé et constituer ainsi la bibliothèque mathématique de calcul symbolico-numérique eXMSL. IMSL FORTRAN Library Version 5.0 propose une large gamme de fonctions de résolution de problèmes numériques écrites en FORTRAN 90 ou en FORTRAN 77. L'appel à ces routines fiables et performantes est naturel depuis un code écrit en FORTRAN 90.

La bibliothèque IMSL permet la réalisation de l'ensemble des services proposés par le système de calcul numérique spécifié dans le diagramme de classes de la Figure 54.

La routine **DLSARG** résout un système linéaire réel avec raffinement itératif au voisinage de la solution, à partir de la donnée de la matrice incidente. La méthode **linearEquationRoot(a : Expression, b : Expression) : LinearEquationRootExpression** appelle la routine **DLSARG** pour proposer un service de résolution directe de systèmes linéaires.

La routine **DG2RES** résout un système linéaire réel par application d'une méthode itérative de résidu minimal généralisé. La méthode **linearEquationRoot(aTimesx : FunctionExpression, b : Expression, initialGuess : Expression) : LinearEquationRootExpression** appelle la routine **DG2RES** pour proposer un service de résolution itérative de systèmes linéaires.

La routine **DNEQNJ** résout un système d'équations non linéaires en utilisant un algorithme hybride de Powell modifié exploitant une matrice Jacobienne analytique. La méthode **nonLinearEquationRoot(residualEvaluation : FunctionExpression, jacobianEvaluation : FunctionExpression, initialGuess : Expression) : NonLinearEquationRootExpression** appelle la routine **DNEQNJ** pour proposer un service de résolution de systèmes non linéaires.

La routine **DD2SPG** résout un système d'équations algébro-différentielles du premier ordre en appliquant l'algorithme de Gear BDF. La méthode **differentialAlgebraicEquationRoot(residualEvaluation : FunctionExpression, jacobianEvaluation : FunctionExpression, dependentVariableStartingValues : Expression, dependentVariableDerivativeStartingValues : Expression, independentVariableValues : Expression) : DifferentialAlgebraicEquationRootExpression** appelle la routine **DD2SPG** pour proposer un service de résolution de systèmes d'équations algébro-différentielles du premier ordre.

La routine **DNLPG** utilise une méthode de programmation quadratique successive sous contraintes égalités pour minimiser un critère sous contraintes inégalités. Les gradients analytiques du critère et des contraintes sont fournis à cette routine. La méthode

minimum(criterionEvaluation : FunctionExpression, criterionDerivativeEvaluation : FunctionExpression, initialGuess : Expression, numberOfConstraints : Expression, numberOfEqualityConstraints : Expression) : MinimumExpression appelle la routine **DNLPG** pour proposer un service de résolution de problèmes de programmation non linéaire.

Les routines de la bibliothèque mathématique IMSL sont appelées à partir des méthodes de la classe **NumericalCalculationSystem**, transposées dans le module **CalculationSystem_**, selon un principe systématique. L'interface de chaque méthode de résolution numérique R_i de la classe **NumericalCalculationSystem** comprend un ou deux paramètres formels du type **FunctionExpression**. Selon la nature du problème à résoudre, ces fonctions $f_{i,1}$, $f_{i,2}$, … permettent l'évaluation d'une fonction résidu, d'un critère ou d'une fonction dérivée en un point quelconque. Les variables de ces fonctions d'évaluation sont du type **Expression** de même que leurs résultats. Si les routines composant une bibliothèque mathématique numérique, IMSL notamment, requièrent des fonctions pour l'évaluation d'une fonction résidu, d'un critère ou d'une fonction dérivée en un point quelconque, ces routines exigent des fonctions dont les variables et les résultats sont des nombres réels, ou des vecteurs de nombres réels, et non des objets du type **Expression**. Par conséquent, lors de la génération du code FORTRAN 90 à partir du modèle d'implémentation, dans chaque fonction R_i sont insérées des fonctions $g_{i,1}$, $g_{i,2}$, …, respectivement associées aux fonctions $f_{i,1}$, $f_{i,2}$, …, et dont le rôle est d'adapter celles-ci à une utilisation par la bibliothèque mathématique numérique utilisée. Plus précisément, le code d'une fonction $g_{i,j}$ associée à la fonction $f_{i,j}$ se décompose en trois parties :

1. conversion des paramètres formels de $g_{i,j}$ dans le système de types proposé dans ce travail : les scalaires sont convertis en variables de type **Expression**, tandis que les vecteurs de nombres sont convertis en tableaux dont les éléments sont du type **ExpressionArrayElement** ;
2. appel de la fonction d'évaluation $f_{i,j}$ avec les arguments convertis ;
3. conversion du résultat d'évaluation de $f_{i,j}$ dans un type primitif du langage FORTRAN 90. Le résultat converti de l'évaluation de $f_{i,j}$ devient le résultat de l'évaluation de $g_{i,j}$.

3.2.2. Composant logiciel de calcul symbolico-numérique

3.2.2.1. Formalisation du contenu des échanges par le dialecte MathML

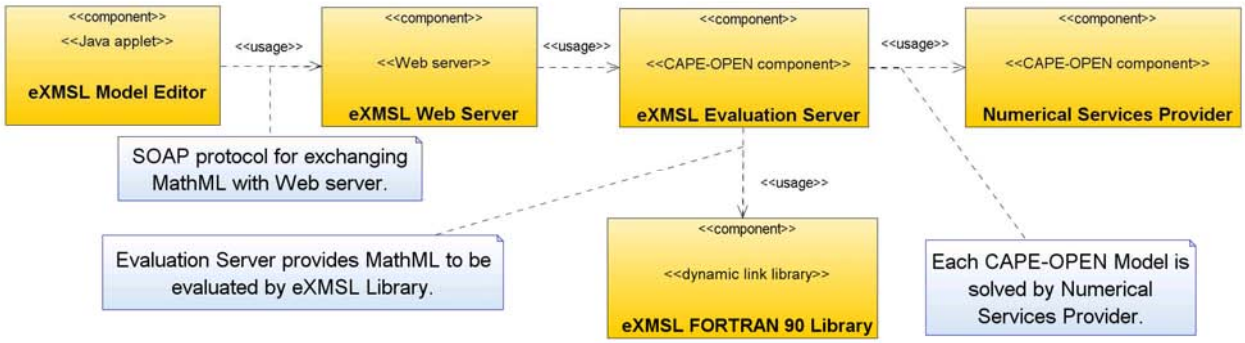

Figure 59 – eXMSL on the Web – diagramme de composants UML.

Le système de calcul symbolico-numérique proposé a également été envisagé dans le cadre d'une architecture standardisée, CAPE-OPEN (Belaud & Braunschweig 2002;Belaud, Braunschweig, & Pons 2002), à base de composants logiciels, dont le but est le dialogue d'éléments issus de sources diverses pour la simulation en ingénierie des procédés. L'architecture résultante définit plusieurs composants logiciels :

- eXMSL Model Editor ;
- eXMSL Web Server ;
- eXMSL Evaluation Server ;
- eXMSL FORTRAN 90 Library ;
- Numerical Services Provider, un composant compatible avec la spécification CAPE-OPEN Numerical Solver.

eXMSL Evaluation Server est en charge de la construction de modèles au niveau des équations, ainsi que de l'évaluation de ces modèles et de leurs dérivées en un point quelconque. Le diagramme de composants de la Figure 59 présente les différents niveaux de l'architecture multi-étagée dans laquelle vient s'insérer ce composant.

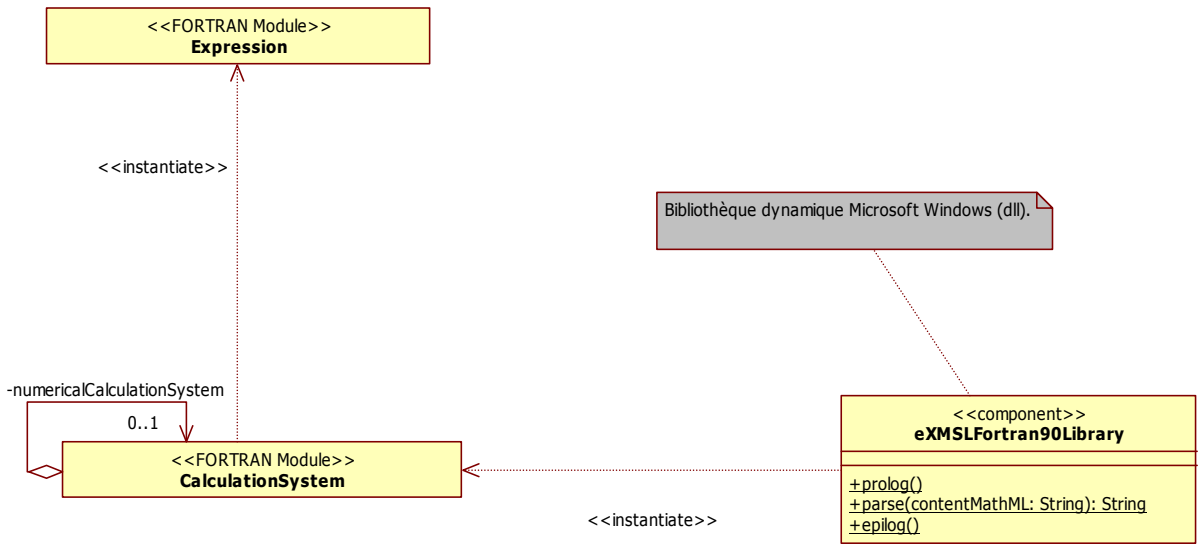

Figure 60 – Modèle d'implémentation de la bibliothèque mathématique symbolico-numérique eXMSL FORTRAN 90 Library : accès à partir d'un document au format MathML de contenu.

Le diagramme de classes de la Figure 60 précise l'interface du composant eXMSL FORTRAN 90 Library, dédié au calcul symbolico-numérique et sur lequel s'appuie eXMSL Evaluation Server. Les descriptions de modèles et les évaluations de modèles sont toutes deux codées dans une application XML : MathML 2.0. Ce format textuel et standardisé a été choisi comme le moyen de produire la représentation graphique de nos modèles et d'assurer l'interopérabilité entre les composants eXMSL. La méthode **parse** ajoutée à la classe **eXMSLFortran90Library** analyse un document MathML et produit une évaluation symbolico-numérique de celui-ci.

3.2.2.2. Formalisation de l'accès aux services de résolution

eXMSL Evaluation Server implémente les spécifications CAPE-OPEN, Equation Set Object et Model, utilisant les technologies Microsoft .NET[24].

eXMSL Evaluation Server peut s'appuyer sur eXMSL FORTRAN 90 Library pour une résolution symbolique et numérique et/ou sur un composant logiciel compatible CAPE-OPEN pour une résolution numérique. Par exemple, Numerical Services Provider est un composant CAPE-OPEN qui résout numériquement tout système d'équations, non linéaires ou algébro-différentielles du premier ordre. (Belaud et al. 2001) décrit ce composant.

[24] Ce composant est en cours de développement, un groupe du CO-LaN (CO-LaN 2007) réfléchissant actuellement à l'intégration de la technologie Microsoft .NET dans le standard CAPE-OPEN (Barrett et al. 2007).

3.2.2.3. Interface graphique d'édition et d'évaluation d'expressions mathématiques

eXMSL Model Editor exploite ces composants métier afin de fournir les services d'interaction utilisateur selon un mode d'accès universel de client riche. Basé sur la technologie applet Java, il permet aux utilisateurs d'éditer, de résoudre et d'optimiser des modèles au travers d'une interface graphique. Cette interface n'est pas une interface orientée blocs, comme peut l'être l'interface de MATLAB/SIMULINK. Travailler avec eXMSL Model Editor à partir d'un navigateur Web revient à dérouler une session de calcul sous Mathematica ou Maple : les expressions d'entrée sont saisies à l'aide d'un éditeur d'équations, toute expression peut être évaluée, produisant un résultat qui lui-même est une nouvelle expression.

L'interface graphique de l'application est présentée dans la Figure 61. Le niveau de présentation d'eXMSL on the Web comprend trois panneaux : le panneau du haut regroupe chaque expression saisie avec le résultat de son évaluation, le panneau du milieu consiste en un éditeur d'équations[25], et le panneau du bas organise les éléments de l'application soit sous la forme de menus déroulants, soit sous la forme de dossiers triés par ordre lexicographique. Un scénario typique d'utilisation d'eXMSL on the Web débute par l'édition d'une expression mathématique à l'aide de l'éditeur d'équations. Cette expression est ensuite évaluée en cliquant sur le bouton `Evaluate`. L'expression en entrée et l'expression en sortie sont ensuite toutes deux affichées dans le panneau du haut, le panneau document. Sur le modèle du système Mathematica, ces expressions sont désignées par les termes `In(n)` et `Out(n)` pour un usage ultérieur. Le panneau dictionnaire, en bas de l'interface graphique, maintient une liste des symboles disponibles -symboles prédéfinis ou définis par l'utilisateur-, ainsi qu'une liste choisie de sous-expressions pouvant être utilisées lors de la construction de nouvelles expressions.

[25] WebEQ applet par Design Science Inc.

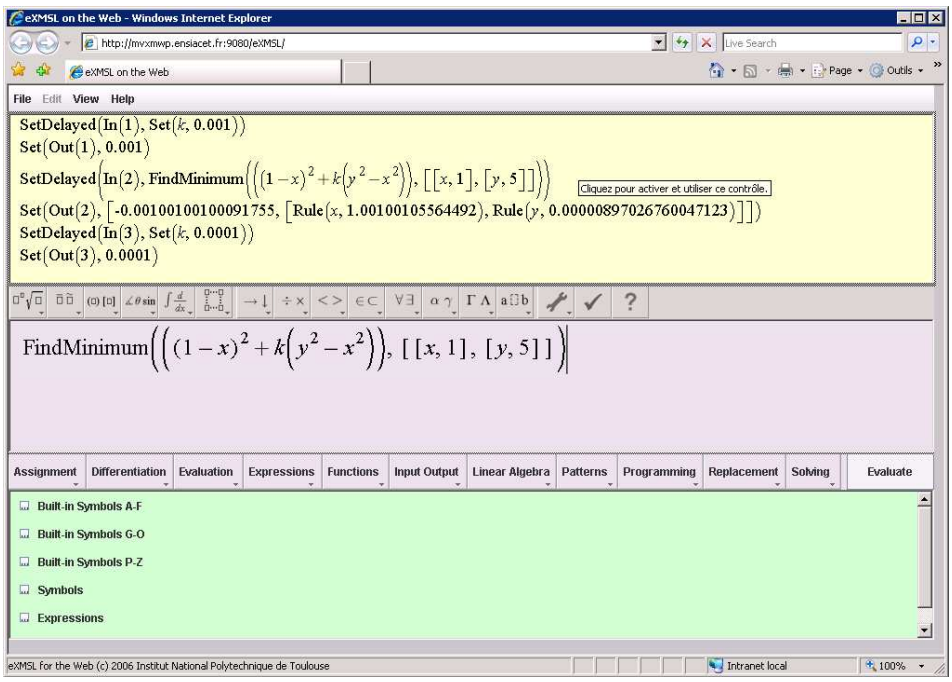

Figure 61 – Minimisation d'un critère à l'aide d'eXMSL on the Web.

La session présentée en Figure 61 tente de minimiser une fonction de deux variables, à partir d'une estimation initiale du minimum. La dérivée symbolique du critère est calculée par le composant eXMSL FORTRAN 90 library en charge des transformations formelles. Un minimum local, éventuellement global, est trouvé et retourné à eXMSL Evaluation Server. L'expression en entrée ainsi que le résultat de l'évaluation sont toutes deux traduites dans le dialecte MathML. Le document résultant est retourné via le protocole SOAP à eXMSL Model Editor pour affichage. Annexe D donne un exemple d'un tel document, correspondant à la minimisation présentée en Figure 61.

3.3. Synthèse

La validation du modèle de conception d'un système de calcul symbolico-numérique fait l'objet du Chapitre 3. En suivant une approche orientée modèle, deux transformations sont formalisées qui conduisent du modèle de conception initial au modèle d'implémentation, puis à une réalisation technique dans le langage FORTRAN 90.

Le prototype logiciel produit se présente sous la forme d'une bibliothèque mathématique acceptant des spécifications de problèmes dans une syntaxe proche du langage mathématique.

A partir de cette bibliothèque mathématique, est bâtie une architecture Web pour un accès universel aux services de calcul proposé.

Chapitre 4. Applications du système de calcul symbolico-numérique proposé

Le système de calcul produit est appliqué à quelques problèmes afin d'analyser les avantages et les inconvénients liés notamment au modèle de coopération choisi. Si ce modèle demeure indépendant du domaine d'application, en relation avec les thématiques abordées par le Laboratoire de Génie Chimique, les études de cas présentées ici appartiennent au domaine scientifique de l'ingénierie des procédés.

La partie 4.1 référence des problèmes ayant servi à la validation du prototype logiciel utilisé. Le modèle de solvatation d'Engels, résolu dans la partie 4.2, illustre la résolution des équations algébriques non linéaires, formulées à l'aide d'évaluations de fonctions implicites. La partie 4.3 détaille un modèle algébro-différentiel de la distillation de Rayleigh, durant laquelle le système de calcul formel eXMSL produit les expressions mathématiques utiles au calcul de conditions initiales cohérentes et à l'intégration numérique.

4.1. Validation

Des modèles de taille réduite, issus de la littérature, ont permis la validation du système de calcul eXMSL. (Alloula, Belaud, & Le Lann 2004) relate la simulation de deux exemples: la combustion du propane dans l'air, conduisant à un système de douze équations non linéaires, et le problème d'équilibre entre phases exprimé comme une minimisation de l'énergie libre de Gibbs-Duhem. (Alloula, Belaud, Leibovici, & Le Lann 2006) décrit le calcul de propriétés thermodynamiques à partir du modèle par équation d'état SRK, où le facteur de compressibilité est une fonction implicite de la température et de la composition du mélange, définie par une équation et deux inéquations algébriques non linéaires. (Alloula, Belaud, & Le Lann 2007) fait référence à un modèle d'évaluation du pH du vin dont la formulation utilise la notion de fonctions implicites définies par des équations algébriques non linéaires.

4.2. Modèle de solvatation d'Engels

4.2.1. Présentation

4.2.1.1. Equilibre de solvatation

Le phénomène de solvatation peut être envisagé comme l'agglomération de molécules de solvant autour d'une autre espèce chimique, l'électrolyte, contribuant ainsi à la formation de molécules complexes. Un tel phénomène peut se traduire par l'équilibre chimique suivant :

$$\nu_S S + E \leftrightarrow \nu_C C$$

Ce genre de modèle est particulièrement adapté à la représentation thermodynamique des solutions aqueuses d'acides forts.

4.2.1.2. Espèces vraies et espèces apparentes

La solution contient, une fois l'équilibre précédent atteint, les espèces S, E et C. Ce sont les espèces réelles, dont les fractions molaires sont respectivement notées x_S, x_E et x_C. Toutefois, seules sont connues les fractions molaires des espèces mises en jeu au début du mélange, dites espèces apparentes. Les fractions molaires apparentes du solvant et de l'électrolyte sont respectivement notées x_S^0 et x_E^0. Dans un simulateur de procédés chimiques, les calculs peuvent faire référence aux espèces apparentes, en exprimant toutes les propriétés thermodynamiques en fonction des fractions molaires de celles-ci. Il est alors intéressant de définir les coefficients d'activité apparents du solvant et de l'électrolyte par les relations :

$$\begin{cases} x_S^0 \cdot \gamma_S^0(T, x_S^0, x_E^0) = x_S \cdot \gamma_S(T, x_S, x_E, x_C) \\ x_E^0 \cdot \gamma_E^0(T, x_S^0, x_E^0) = x_E \cdot \gamma_E(T, x_S, x_E, x_C) \end{cases}$$

Équation 19 - Définition des coefficients d'activité apparents dans le modèle de solvatation d'Engels.

Ces définitions permettent de formuler les équations caractéristiques des différents équilibres thermodynamiques à partir des coefficients d'activité apparents, donc sans faire apparaître explicitement les espèces réelles. Les mêmes modèles servent dès lors à calculer les propriétés d'équilibre d'un mélange, qu'il y ait solvatation ou non.

4.2.2. Calcul des coefficients d'activité apparents et de leurs dérivées

La spécificité du modèle de solvatation introduit dans (Engels 1990) réside dans le fait que les fractions molaires réelles sont des fonctions implicites des fractions molaires apparentes, obtenues par résolution du système de trois équations non linéaires à trois inconnues :

$$\begin{cases} x_S = x_S^0 + \dfrac{x_C}{\nu_C} \cdot \left(x_S^0 \cdot (\nu_S + 1 - \nu_C) - \nu_S \right) \\ x_E = x_E^0 + \dfrac{x_C}{\nu_C} \cdot \left(x_E^0 \cdot (\nu_S + 1 - \nu_C) - 1 \right) \\ K_a(T) = \dfrac{(x_C \cdot \gamma_C(T, x_S, x_E, x_C))^{\nu_C}}{(x_S \cdot \gamma_S(T, x_S, x_E, x_C))^{\nu_S} \cdot (x_E \cdot \gamma_E(T, x_S, x_E, x_C))} \end{cases}$$

Équation 20 – Définition des fractions molaires réelles à partir des fractions molaires apparentes dans le modèle de solvatation d'Engels.

La première équation exprime la conservation de l'espèce S. La seconde équation exprime la conservation de l'espèce E. La dernière équation traduit l'équilibre de complexation dont $K_a(T)$ est la constante. L'expression analytique explicite des coefficients d'activité réels γ_S, γ_E et γ_C est choisie parmi les différents modèles : Wilson, NRTL, etc.

Les coefficients d'activité apparents sont définis à partir des fractions molaires réelles, elles-mêmes fonctions implicites des fractions molaires apparentes. Il est donc la plupart du temps impossible d'obtenir une expression analytique explicite des coefficients d'activité apparents en fonction des fractions molaires apparentes. A fortiori il est impossible d'obtenir une expression analytique explicite des dérivées des coefficients d'activité apparents, dérivées requises notamment lors du calcul de propriétés d'équilibre. La résolution numérique ou symbolico-numérique de Équation 20 est facilitée par une estimation des fractions molaires ayant un sens physique : celles-ci sont initialisées à la moyenne des valeurs correspondant respectivement à une dissociation complète et à l'absence de dissociation.

4.2.2.1. Evaluation numérique

L'évaluation numérique des coefficients d'activité apparents $\gamma_S^0(T, x_S^0, x_E^0)$ et $\gamma_E^0(T, x_S^0, x_E^0)$, en un point (T, x_S^0, x_E^0) donné, comprend deux étapes :

1. le calcul des valeurs des fractions molaires réelles au point (T, x_S^0, x_E^0), soit $x_s(T, x_S^0, x_E^0)$, $x_s(T, x_S^0, x_E^0)$ et $x_s(T, x_S^0, x_E^0)$, par résolution numérique du système d'équations non linéaires précédent ;
2. l'application des définitions
$$\gamma_S^0(T, x_S^0, x_E^0) = \frac{x_S(T, x_S^0, x_E^0)}{x_S^0} \cdot \gamma_S(T, x_S(T, x_S^0, x_E^0), x_E(T, x_S^0, x_E^0), x_C(T, x_S^0, x_E^0))$$
$$\text{et } \gamma_E^0(T, x_S^0, x_E^0) = \frac{x_E(T, x_S^0, x_E^0)}{x_E^0} \cdot \gamma_E(T, x_S(T, x_S^0, x_E^0), x_E(T, x_S^0, x_E^0), x_C(T, x_S^0, x_E^0)).$$

L'évaluation numérique des dérivées des coefficients d'activité apparents $\gamma_S^{0\prime}(T, x_S^0, x_E^0)$ et $\gamma_E^{0\prime}(T, x_S^0, x_E^0)$, en un point (T, x_S^0, x_E^0) donné, utilise la plupart du temps un schéma aux

différences finies. Lorsque le schéma aux différences finies est d'ordre un, l'évaluation numérique des quantités suivantes est requise :

$\gamma_S^0(T,x_S^0,x_E^0)$, $\gamma_E^0(T,x_S^0,x_E^0)$, $\gamma_S^0(T+\delta T,x_S^0,x_E^0)$, $\gamma_E^0(T+\delta T,x_S^0,x_E^0)$, $\gamma_S^0(T,x_S^0+\delta x_S^0,x_E^0)$, $\gamma_E^0(T,x_S^0+\delta x_S^0,x_E^0)$, $\gamma_S^0(T,x_S^0,x_E^0+\delta x_E^0)$ et $\gamma_E^0(T,x_S^0,x_E^0+\delta x_E^0)$. Le calcul approché de la matrice jacobienne $\begin{bmatrix}\gamma_S^{0\prime}(T,x_S^0,x_E^0)\\ \gamma_E^{0\prime}(T,x_S^0,x_E^0)\end{bmatrix}$ comprend alors trois séquences de deux étapes supplémentaires par rapport à l'évaluation numérique de $\begin{bmatrix}\gamma_S^0(T,x_S^0,x_E^0)\\ \gamma_E^0(T,x_S^0,x_E^0)\end{bmatrix}$: une séquence pour l'évaluation de $\begin{bmatrix}\gamma_S^0(T+\delta T,x_S^0,x_E^0)\\ \gamma_E^0(T+\delta T,x_S^0,x_E^0)\end{bmatrix}$, une séquence pour l'évaluation de $\begin{bmatrix}\gamma_S^0(T,x_S^0+\delta x_S^0,x_E^0)\\ \gamma_E^0(T,x_S^0+\delta x_S^0,x_E^0)\end{bmatrix}$, et enfin une séquence pour l'évaluation de $\begin{bmatrix}\gamma_S^0(T,x_S^0,x_E^0+\delta x_E^0)\\ \gamma_E^0(T,x_S^0,x_E^0+\delta x_E^0)\end{bmatrix}$.

4.2.2.2. Evaluation par le système de calcul symbolico-numérique proposé

Une alternative au calcul des dérivées des coefficients d'activité apparents par différences finies est leur calcul par application d'un théorème des fonctions implicites, tel que présenté en 2.3.1. Plus précisément, ce théorème peut être appliqué au calcul de la dérivée des fractions molaires réelles, la dérivée des coefficients d'activité apparents étant ensuite évaluée par application de la formule de dérivation des fonctions composées. Les deux avantages attendus de cette approche sont : une fiabilité accrue des dérivées calculées et un gain en temps de calcul.

De façon générale, le calcul des dérivées par différences finies pose le problème du choix de l'amplitude de la perturbation des variables. Dans ce cas particulier, à cette question se rajoute la question du choix du critère d'arrêt lors de la résolution du système d'équations non linéaires Équation 20. L'approche symbolico-numérique supprime ces choix. La résolution directe des systèmes linéaires de petite taille (3x3) ne devrait pas introduire des difficultés numériques nouvelles.

La résolution de trois systèmes de trois équations linéaires, partageant de plus la même matrice incidente, devrait être plus rapide que la résolution de trois systèmes de trois équations non linéaires requis par la méthode de différences finies.

Le mélange choisi pour cette étude est une solution aqueuse d'acide iodidrique (HI) pour laquelle l'équilibre de solvatation est le suivant :

$$5H_2O + HI \leftrightarrow 2C$$

Les coefficients d'activité sont calculés à partir du modèle de Wilson. L'expression de la constante d'équilibre de complexation, ainsi que les coefficients d'interaction binaire sont issus de (Engels 1990).

Le modèle est constitué par un ensemble de modules FORTRAN 90, tirant parti des services fournis par la bibliothèque symbolico-numérique eXMSL FORTRAN 90 Library. Le système de calcul numérique utilisé dans cette étude est la bibliothèque mathématique IMSL, chargée de résoudre les systèmes linéaires après factorisation de la matrice incidente.

Selon le système de types de fonctions détaillé dans la partie 2.3, les fractions molaires réelles sont représentées par une fonction implicite x qui, pour chaque valeur de (T, x_S^0, x_E^0), calcule (x_S, x_E, x_C) par résolution du système d'équations non linéaires Équation 20. Les coefficients d'activité réels sont représentés par la composition de deux fonctions : le modèle de coefficient d'activité choisi, et la fonction explicite[26] définie par $(T, x_S^0, x_E^0) \mapsto (T, x_S, x_E, x_C)$. Les coefficients d'activité apparents sont représentés par la composition de deux fonctions : la fonction explicite définie par $(x_S^0, x_E^0, x_S, x_E, x_C, \gamma_S, \gamma_E, \gamma_C) \mapsto (\frac{x_S}{x_S^0} \cdot \gamma_S, \frac{x_E}{x_E^0} \cdot \gamma_E)$ et la fonction explicite définie par

$$(T, x_S^0, x_E^0) \mapsto (x_S^0, x_E^0, x_S(T, x_S^0, x_E^0), x_E(T, x_S^0, x_E^0), x_C(T, x_S^0, x_E^0), \gamma_S(T, x_S^0, x_E^0), \gamma_E(T, x_S^0, x_E^0), \gamma_C(T, x_S^0, x_E^0))$$

.

La mise en forme du problème dans le cadre du système de calcul symbolico-numérique proposé a donc fait un usage systématique de l'approche fonctionnelle, naturelle dans un système de calcul formel. Les entités modélisées dans le modèle structurel de système de calcul formel - fonctions explicites ou implicites, composition de fonctions, matrices Jacobiennes, dérivées directionnelles- sont toutes utilisées lors de la spécification du problème, ou en cours de résolution. Le système de calcul numérique intervient lors de l'évaluation numérique de la fonction implicite x (résolution d'un système non linéaire), et lors du calcul des dérivées de cette fonction (résolution d'un ensemble de systèmes linéaires).

Le traitement de cette étude préalable par le système de calcul symbolico-numérique proposé apporte déjà plusieurs enseignements intéressants.

---

[26] La fonction définie par $(T, x_S^0, x_E^0) \mapsto (T, x_S, x_E, x_C)$ est considérée comme explicite. Pourtant trois de ses quatre fonctions coordonnées sont calculées par évaluation d'une fonction implicite. La continuité entre les notions de fonction implicite et de fonction explicite justifie dès lors le traitement uniforme des fonctions dans le système de calcul symbolico-numérique proposé.

Tout d'abord, l'approche coopérative adoptée où deux systèmes de calcul, l'un formel, l'autre numérique, collaborent intimement à la résolution d'un problème a été appliquée.

L'objectif de fiabilité fixé au système de calcul symbolico-numérique conçu au Chapitre 2 a été atteint ici.

Conditions : Pression (atm)	Conditions : Température (℃)	Mélange (Molaire) : WATER	Mélange (Molaire) : HYDROGEN IODIDE	Ratio (Molaire) : Taux de vaporisation	Coefficients d'activité : WATER	Coefficients d'activité : HYDROGEN IODIDE	Fractions liquide (Molaire) : WATER	Fractions liquide (Molaire) : HYDROGEN IODIDE
50	150	1	0	0,0000	1,0000	0,0000	1,0000	0,0000
50	150	0,95	0,05	0,0000	0,9291	0,0001	0,9500	0,0500
50	150	0,9	0,1	0,0000	0,7621	0,0008	0,9000	0,1000
50	150	0,85	0,15	0,0000	0,4725	0,0220	0,8500	0,1500
50	150	0,8	0,2	0,0000	0,2919	0,2274	0,8000	0,2000
50	150	0,75	0,25	0,0000	0,2324	0,5075	0,7500	0,2500
50	150	0,7	0,3	0,0000	0,2050	0,7101	0,7000	0,3000
50	150	0,65	0,35	0,0000	0,1894	0,8379	0,6500	0,3500
50	150	0,6	0,4	0,0000	0,1798	0,9147	0,6000	0,4000
50	150	0,55	0,45	0,0000	0,1737	0,9593	0,5500	0,4500
50	150	0,5	0,5	0,0000	0,1698	0,9839	0,5000	0,5000
50	150	0,45	0,55	0,0000	0,1675	0,9961	0,4500	0,5500
50	150	0,4	0,6	0,0000	0,1664	1,0010	0,4000	0,6000
50	150	0,35	0,65	0,1091	0,1663	1,0014	0,3921	0,6079
50	150	0,3	0,7	0,2386	0,1663	1,0014	0,3921	0,6079
50	150	0,25	0,75	0,3682	0,1663	1,0014	0,3921	0,6079
50	150	0,2	0,8	0,4977	0,1663	1,0014	0,3921	0,6079
50	150	0,15	0,85	0,6272	0,1663	1,0014	0,3921	0,6079
50	150	0,1	0,9	0,7568	0,1663	1,0014	0,3921	0,6079
50	150	0,05	0,95	0,8863	0,1663	1,0014	0,3921	0,6079

Tableau 2 – Coefficients d'activité d'un mélange H_2O, HI calculés par Simulis Thermodynamics (T=150℃).

Le Tableau 2 affiche les valeurs des coefficients d'activité apparents obtenues par le logiciel commercial Simulis Thermodynamics (Vacher & Guittard 2002) à une température de 150 degrés Celsius pour différentes compositions molaires initiales. Au-delà d'une certaine fraction molaire de l'acide, comprise entre 0,45 et 0,5, et pour une valeur de la pression égale à cinquante atmosphères, le mélange devient diphasique, et les fractions molaires apparentes en phase liquide demeurent inchangées.

				eXMSL		Simulis Thermodynamics	
Pression (atm)	Température (℃)	Mélange (Molaire) : WATER	Mélange (Molaire) : HYDROGEN IODIDE	Coefficients d'activité : WATER	Coefficients d'activité : HYDROGEN IODIDE	Coefficients d'activité : WATER	Coefficients d'activité : HYDROGEN IODIDE
50	150	1	0	1,00000	0,00000	1,00000	0,00003
50	150	0,95	0,05	0,92914	0,00007	0,92914	0,00007
50	150	0,9	0,1	0,76208	0,00081	0,76208	0,00081
50	150	0,85	0,15	0,47251	0,02204	0,47251	0,02204
50	150	0,8	0,2	0,29193	0,22739	0,29193	0,22739
50	150	0,75	0,25	0,23240	0,50751	0,23240	0,50751
50	150	0,7	0,3	0,20496	0,71014	0,20496	0,71014
50	150	0,65	0,35	0,18944	0,83787	0,18944	0,83787
50	150	0,6	0,4	0,17981	0,91472	0,17981	0,91472
50	150	0,55	0,45	0,17365	0,95932	0,17365	0,95932
50	250	1	0	1,00000	0,00000	1,00000	0,00004
50	250	0,95	0,05	0,92309	0,00043	0,92309	0,00043
50	250	0,9	0,1	0,74846	0,00563	0,74846	0,00563
50	250	0,85	0,15	0,51774	0,07540	0,51774	0,07540
50	250	0,8	0,2	0,38915	0,30031	0,38915	0,30031

Tableau 3 – Coefficients d'activité d'un mélange H_2O, HI calculés par eXMSL et Simulis Thermodynamics (T=150℃ et T=250℃).

La dépendance des paramètres d'interaction binaire vis-à-vis de la température, l'expression de la constante d'équilibre de complexation, ainsi que l'expression de la pression de vapeur saturante de HI sont identiques dans eXMSL, Simulis Thermodynamics et dans le code de calcul développé par Engels. eXMSL et Simulis Thermodynamics adoptent la même expression de la vapeur saturante de l'eau, tandis qu'Engels utilise une expression empruntée à la littérature mais que nous n'avons pas pu retrouver. Les valeurs des coefficients d'activité apparents obtenus à l'aide d'eXMSL et de Simulis Thermodynamics dans les mêmes conditions opératoires, et pour des fractions molaires apparentes pour lesquelles le mélange est monophasique, sont inventoriées dans le Tableau 3. Les résultats obtenus par les deux environnements de calcul sont parfaitement identiques à la précision choisie.

La comparaison des dérivées des coefficients d'activité apparents obtenus par eXMSL avec ceux calculés par Simulis Thermodynamics est par contre impossible, ces grandeurs n'étant pas accessibles à l'utilisateur de ce simulateur. La validation des matrices jacobiennes, calculées de façon symbolico-numérique, a été obtenue par comparaison avec les résultats de l'application aux fonctions modélisant les coefficients d'activité apparents d'un schéma aux différences finies. Le calcul par eXMSL de dérivées directionnelles selon les vecteurs de base de $\Re^3$ a restitué très exactement les vecteurs colonnes des matrices jacobiennes calculées précédemment, prouvant la cohérence des deux types de calcul. De plus le calcul de la matrice jacobienne de la fonction définie par $(T, x_S^0, x_E^0) \mapsto (\ln(\gamma_s^0(T, x_S^0, x_E^0)), \ln(\gamma_E^0(T, x_S^0, x_E^0)))$ a permis de vérifier très précisément l'identité de Gibbs-Duhem $\frac{\partial \ln(\gamma_s^0(T, x_S^0, x_E^0))}{\partial x_E^0} = \frac{\partial \ln(\gamma_E^0(T, x_S^0, x_E^0))}{\partial x_S^0}$, valide dans le cas particulier d'un binaire.

L'objectif de performance fixé au système de calcul symbolico-numérique conçu au Chapitre 1 n'a par contre pas été atteint ici. Sans avoir à faire de mesure précise, le calcul par eXMSL des coefficients d'activité apparents de l'eau et de l'acide iodidrique, à une température donnée et pour cent compositions initiales différentes du mélange, est nettement plus lent que le même calcul fait par Simulis Thermodynamics, qui lui apparaît immédiat. Cette différence est à attribuer au coût des étapes de transformation formelle que sont le calcul de la fonction résidu associée à Équation 20, l'obtention de la dérivée exacte de cette fonction, la composition des fonctions intervenant dans le problème et les simplifications algébriques des différentes expressions manipulées. Le temps de calcul par eXMSL des dérivées des coefficients d'activité apparents de l'eau et de l'acide iodidrique ne peut pas être directement comparé au temps de calcul de ces mêmes dérivées par Simulis Thermodynamics puisque ce résultat n'est pas directement accessible à l'utilisateur.

Des possibilités de réutilisation par composition de modèles peuvent déjà être envisagées dans le cadre du système de calcul symbolico-numérique proposé. Le modèle de coefficients d'activité apparents utilisé pour l'étude du binaire (H_2O, HI) est générique : le modèle de coefficient d'activité de Wilson, représenté par une fonction, peut aisément être remplacé par un autre modèle de coefficient d'activité ; de même le binaire (H_2O, HI) peut être aisément remplacé par un autre binaire, caractérisé par sa stoechiométrie et ses paramètres d'interaction binaire. Ce modèle générique de coefficients d'activité apparents peut être composé avec d'autres modèles, tels que les modèles d'équilibre thermodynamique, pour modéliser des systèmes physico-chimiques. La composition de modèles prend alors deux formes, déjà utilisées lors de cette modélisation :

1. la composition de fonctions ;
2. la définition de fonctions comprenant des évaluations de fonctions précédemment définies.

Le traitement de cette étude préalable nous a confronté à une difficulté non prise en compte lors de la conception du système de calcul symbolico-numérique. Cette difficulté est liée à l'exigence de définir des fonctions, comprenant des évaluations de fonctions précédemment définies et à valeurs dans un espace produit ($\Re^n$ avec $n \geq 2$). La fonction $\tilde{x} : (T, x_S^0, x_E^0) \mapsto (T, x_S, x_E, x_C)$ en est un exemple. Une définition plus précise de celle-ci est la suivante : $\tilde{x} : (T, x_S^0, x_E^0) \mapsto (T, x(T, x_S^0, x_E^0))$, en désignant par x la fonction implicite définie de $\Re^3$ dans $\Re^3$ à l'aide de Équation 20. La définition de $\tilde{x}$ est construite en utilisant une évaluation de la fonction x, précédemment définie et à valeurs dans $\Re^3$. Calculer la matrice jacobienne de la fonction $\tilde{x}$ à l'aide de l'opérateur de dérivation partielle[27] proposé par tous les systèmes de calcul formel, donnerait un résultat faux dans ce cas. En adoptant la syntaxe de Mathematica, la matrice retournée par un système de calcul, ignorant la notion de fonction implicite, serait le résultat de l'évaluation de l'expression $\begin{pmatrix} D[T,T] & D[T,x_S^0] & D[T,x_E^0] \\ D[x[T,x_S^0,x_E^0],T] & D[x[T,x_S^0,x_E^0],x_S^0] & D[x[T,x_S^0,x_E^0],x_E^0] \end{pmatrix}$, c'est-à-dire une matrice à deux lignes et trois colonnes. Utiliser uniquement l'opérateur de dérivation partielle, tel que c'est le cas dans les systèmes de calcul formel, n'est pas suffisant : il faut construire la matrice jacobienne d'une fonction par assemblage de termes élémentaires -dérivées partielles de fonctions coordonnées par rapport aux différentes variables-, et de matrices jacobiennes -

[27] Mathematica désigne par **D** l'opérateur de dérivation symbolique. Maple désigne par **diff** celui-ci.

dérivées des fonctions intervenant dans la définition de la fonction initiale. Ce processus de construction de la matrice jacobienne par assemblage est exactement la démarche adoptée par le numéricien lorsqu'il souhaite disposer dans un code de calcul de la dérivée analytique d'une fonction dont les fonctions coordonnées sont des évaluations d'autres fonctions. En désignant par p_1 la projection définie par $p_1 : \begin{array}{c} \Re^3 \to \Re \\ (T, x_S^0, x_E^0) \mapsto T \end{array}$, la fonction $\tilde{x}$ précédente peut être définie par $\tilde{x} : (T, x_S^0, x_E^0) \mapsto (p_1(T, x_S^0, x_E^0), x(T, x_S^0, x_E^0))$, et sa fonction dérivée en un point (T, x_S^0, x_E^0), identifiée à la matrice par blocs $\begin{pmatrix} p_1'(T, x_S^0, x_E^0) \\ x'(T, x_S^0, x_E^0) \end{pmatrix}$. Tous les éléments de cette matrice sont connus précisément si et seulement si $p_1'(T, x_S^0, x_E^0)$ et $x'(T, x_S^0, x_E^0)$ peuvent tous deux être évalués numériquement. x étant une fonction implicite, cette évaluation est possible lorsque T, x_S^0 et x_E^0 ont tous trois une valeur numérique. Cet exemple simple montre tout l'intérêt de pouvoir assembler des matrices jacobiennes lors de la dérivation de fonctions.

Le système de calcul symbolico-numérique proposé automatise non seulement l'étape de dérivation symbolique, mais aussi celle d'assemblage des sous-matrices jacobiennes. Les systèmes de calcul formel traitent évidemment l'étape de dérivation symbolique, mais ils n'abordent jamais la question de l'assemblage des sous-matrices constituant une matrice jacobienne. La raison en est simple : manipulant uniquement des fonctions explicites, la matrice jacobienne est obtenue par dérivation partielle de chaque fonction coordonnée, dont l'expression est connue, par rapport à chaque variable. L'introduction des fonctions implicites, dont une expression algébrique des fonctions coordonnées demeure inconnue ou n'existe pas, impose la prise en charge des matrices par blocs afin de proposer un calcul correct des matrices jacobiennes, ou des dérivées directionnelles.

Dans le but de formaliser la définition de fonctions comprenant des évaluations de fonctions précédemment définies, un nouvel opérateur A, dit opérateur d'assemblage fonctionnel, est défini. Etant données $f_1 : \Re^n \to \Re^{m_1}$, $f_2 : \Re^n \to \Re^{m_2}$, …, $f_p : \Re^n \to \Re^{m_p}$ des fonctions toutes définies sur $\Re^n$, l'application de l'opérateur d'assemblage A au point $(f_1, f_2, ..., f_p)$ construit une nouvelle fonction $A(f_1, f_2, ..., f_p)$ définie par $A(f_1, f_2, ..., f_p) : \begin{array}{c} \Re^n \to \Re^{m_1+m_2+...+m_p} \\ (x_1, ..., x_n) \mapsto \alpha\big(f_1(x_1, ..., x_n), f_2(x_1, ..., x_n), ..., f_n(x_1, ..., x_n)\big) \end{array}$. De façon conjointe est défini un nouvel opérateur α, dit opérateur d'assemblage vectoriel. Etant donnés $V_1 \in \Re^{m_1}$, $V_2 \in \Re^{m_2}$, …, $V_p \in \Re^{m_p}$ des vecteurs réels, l'application de l'opérateur d'assemblage α au point

$(V_1, V_2, ..., V_p)$ construit un nouveau vecteur $\alpha(V_1, V_2, ..., V_p)$ de $\Re^{m_1+m_2+...+m_p}$. Si les coordonnées de chacun des vecteurs V_1, V_2, …, V_p sont connues explicitement alors l'évaluation par le système de calcul symbolico-numérique proposé retourne le vecteur obtenu par concaténation des coordonnées des différents vecteurs, soit $(V_{1,1}, ..., V_{1,m_1}, V_{2,1} ..., V_{2,m_2}, ..., V_{p,1}, ..., V_{p,m_p})$. Dans le cas contraire, le système de calcul retourne l'expression $\alpha(V_1, V_2, ..., V_p)$ inchangée. L'opérateur d'assemblage vectoriel α est également utilisé comme opérateur d'assemblage matriciel, les matrices réelles étant vues comme des vecteurs lignes. Etant données $\mathrm{M}_1 \in \Re^{m_1} \times \Re^n$, $\mathrm{M}_2 \in \Re^{m_2} \times \Re^n$, …, $\mathrm{M}_p \in \Re^{m_p} \times \Re^n$ des matrices réelles comportant toutes n colonnes, l'application de l'opérateur d'assemblage α au point $(\mathrm{M}_1, \mathrm{M}_2, ..., \mathrm{M}_p)$ construit une nouvelle matrice $\alpha(\mathrm{M}_1, \mathrm{M}_2, ..., \mathrm{M}_p) \in \Re^{m_1+m_2+...+m_p} \times \Re^n$. Si les termes de chacune des matrices M_1, M_2, …, M_p sont connus explicitement alors l'évaluation par le système de calcul symbolico-numérique proposé retourne la matrice obtenue par concaténation des lignes des différentes matrices. Dans le cas contraire, le système de calcul retourne l'expression $\alpha(\mathrm{M}_1, \mathrm{M}_2, ..., \mathrm{M}_p)$ inchangée.

L'introduction de l'opérateur d'assemblage vectoriel α permet de disposer, dans le système de calcul symbolico-numérique proposé, de règles pour l'évaluation et la dérivation des fonctions définies par assemblage fonctionnel de fonctions précédemment définies. Équation 21 rappelle la définition d'un assemblage fonctionnel à partir d'un assemblage vectoriel. Équation 22 exprime que la matrice jacobienne d'un assemblage fonctionnel est obtenue par assemblage matriciel des matrices jacobiennes des différentes fonctions. Équation 23 exprime que la dérivée directionnelle d'un assemblage fonctionnel est obtenue par assemblage vectoriel des dérivées directionnelles des différentes fonctions.

$$A(f_1, f_2, ..., f_p)(x_1, x_2, ..., x_n) = \alpha(f_1(x_1, x_2, ..., x_n), f_2(x_1, x_2, ..., x_n), ..., f_p(x_1, x_2, ..., x_n))$$

Équation 21 – Définition d'un assemblage de fonctions.

$$A(f_1, f_2, ..., f_p)'(x_1, x_2, ..., x_n) = \alpha(f_1'(x_1, x_2, ..., x_n), f_2'(x_1, x_2, ..., x_n), ..., f_p'(x_1, x_2, ..., x_n))$$

Équation 22 - Calcul de la matrice jacobienne d'un assemblage de fonctions.

$$\left\lfloor A(f_1, f_2, ..., f_p)'(x_1, x_2, ..., x_n) \right\rfloor \cdot (d_1, d_2, ..., d_n) =$$
$$\alpha(f_1'(x_1, x_2, ..., x_n) \cdot (d_1, d_2, ..., d_n), f_2'(x_1, x_2, ..., x_n) \cdot (d_1, d_2, ..., d_n), ..., f_p'(x_1, x_2, ..., x_n) \cdot (d_1, d_2, ..., d_n))$$

Équation 23 – Calcul de la dérivée directionnelle d'un assemblage de fonctions.

Des règles supplémentaires sont appliquées afin de garantir la correction des évaluations de fonctions et de leurs dérivées lorsque les arguments sont un assemblage[28].

L'utilisation conjointe de l'assemblage et de la composition de fonctions pour construire des modèles constitués de fonctions, explicites ou implicites, semble une voie intéressante de réutilisation des modèles dans le système de calcul proposé. Un parallèle intéressant entre les notions fonctionnelles utilisées ici et les modélisations de systèmes à partir de diagrammes peut être fait :

1. une boîte d'un diagramme causal correspond à une fonction explicite dans le système de calcul symbolico-numérique proposé, ou à une fonction implicite pour laquelle le nombre d'inconnues du système d'équations à résoudre est égal au nombre de fonctions coordonnées ;
2. des boîtes en série correspondent à des fonctions composées ;
3. des boîtes en parallèle ayant la même entrée correspondent à un assemblage de fonctions tel qu'il est défini ci-dessus ;
4. enfin, un réseau cyclique dans un diagramme correspond à une fonction implicite pour laquelle le nombre d'inconnues du système d'équations à résoudre est supérieur au nombre de fonctions coordonnées.

4.2.3. Calcul des propriétés d'équilibre d'un acide fort en solution aqueuse

Le calcul des coefficients d'activité et de leurs dérivées dans le cas d'une solution obéissant au modèle de solvatation d'Engels est appliqué pour déterminer certaines propriétés d'équilibre de l'acide iodhydrique en solution aqueuse : la pression de bulle, la pression de rosée, la température de bulle et la température de rosée.

En utilisant les fractions molaires apparentes du solvant et de l'électrolyte, ainsi que les coefficients d'activité apparents, la pression de bulle P_B peut être vue comme une fonction implicite $P_B : \begin{array}{c} \Re^3 \to \Re \\ (T, x_S^0, x_E^0) \mapsto P_B(T, x_S^0, x_E^0) \end{array}$, où $P_B(T, x_S^0, x_E^0)$ est obtenu par résolution du système non linéaire de trois équations à trois inconnues suivant :

[28] Si x ou d sont des assemblages, f et g des fonctions, l'évaluation des expressions $f(x)$, $g \circ f(x)$, $f'(x)$ et $f'(x) \cdot d$ retourne l'expression initiale.

$$\forall (P_B(T,x_S^0,x_E^0),y_S,y_E)\in\Re^3;\begin{cases} y_S\cdot P_B(T,x_S^0,x_E^0)=\gamma_S^0(T,x_S^0,x_E^0)\cdot x_S^0\cdot P_S^0(T)\\ y_E\cdot P_B(T,x_S^0,x_E^0)=\gamma_E^0(T,x_S^0,x_E^0)\cdot x_E^0\cdot P_E^0(T)\\ x_S^0+x_E^0=y_S+y_E\end{cases}$$

Équation 24 – Calcul de la pression de bulle d'un acide fort en solution aqueuse selon le modèle de solvatation d'Engels.

$P_B(T,x_S^0,x_E^0)$ a en fait une expression explicite, obtenue par sommation des deux premières équations : $P_B(T,x_S^0,x_E^0)=\gamma_S^0(T,x_S^0,x_E^0)\cdot x_S^0\cdot P_S^0(T)+\gamma_E^0(T,x_S^0,x_E^0)\cdot x_E^0\cdot P_E^0(T)$. Nous choisissons d'ignorer celle-ci et de conserver une formulation semblable à celle du calcul de la pression de rosée.

En utilisant les fractions molaires apparentes du solvant et de l'électrolyte, ainsi que les coefficients d'activité apparents, la pression de rosée P_R peut être vue comme une fonction implicite $P_R : \begin{array}{c}\Re^3\to\Re\\ (T,x_S^0,x_E^0)\mapsto P_R(T,x_S^0,x_E^0)\end{array}$, où $P_R(T,x_S^0,x_E^0)$ est obtenu par résolution du système non linéaire de trois équations à trois inconnues suivant :

$$\forall (P_R(T,x_S^0,x_E^0),x_S^0,x_E^0)\in\Re^3;\begin{cases} y_S\cdot P_R(T,x_S^0,x_E^0)=\gamma_S^0(T,x_S^0,x_E^0)\cdot x_S^0\cdot P_S^0(T)\\ y_E\cdot P_R(T,x_S^0,x_E^0)=\gamma_E^0(T,x_S^0,x_E^0)\cdot x_E^0\cdot P_E^0(T)\\ x_S^0+x_E^0=y_S+y_E\end{cases}$$

Équation 25 – Calcul de la pression de rosée d'un acide fort en solution aqueuse selon le modèle de solvatation d'Engels.

Les pressions de bulle et de rosée sont évaluées pour un mélange dont la composition et la température varient. Les conditions opératoires sont celles recensées dans (Engels 1990). Les valeurs d'initialisation fournies au processus de résolution ont un sens physique : les fractions molaires de la phase inconnue sont initialisées aux valeurs des fractions molaires de la phase dont la composition est connue, la pression est approchée par la pression d'un mélange idéal : $P=x_S^0\cdot P_S^0(T)+x_E^0\cdot P_E^0(T)$ ou $P=y_S\cdot P_S^0(T)+y_E\cdot P_E^0(T)$.

			Engels	eXMSL		Simulis Thermodynamics	
Conditions : Température (℃)	Mélange (Molaire) : WATER	Mélange (Molaire) : HYDROGEN IODIDE	Equilibres : Pression de bulle (bar)	Equilibres : Pression de bulle (bar)	Equilibres : Pression de rosée (bar)	Equilibres : Pression de bulle (bar)	Equilibres : Pression de rosée (bar)
107,1	0,983	0,017	1,2555	1,2525	0,6397	1,2525	0,6399
132,4	0,983	0,017	2,8000	2,7991	1,5228	2,7991	1,5272
157,3	0,983	0,017	5,5850	5,5670	3,1732	5,5670	3,1957
170,7	0,983	0,017	7,8030	7,7766	4,5223	7,7766	4,5675
181,9	0,983	0,017	10,1300	10,1119	5,9653	10,1119	6,0415
206,4	0,983	0,017	17,1900	17,1376	10,3573	17,1377	10,5660
230,7	0,983	0,017	27,4600	27,3866	16,7912	27,3866	17,2842
254,9	0,983	0,017	41,8300	41,7667	25,7290	41,7668	26,7762
267,2	0,983	0,017	51,0400	51,0027	31,3817	51,0029	32,8689
90,7	0,965	0,035	0,6645	0,6643	0,3088	0,6643	0,3085
95,7	0,965	0,035	0,8008	0,8003	0,3786	0,8003	0,3783
109,5	0,965	0,035	1,3020	1,3004	0,6424	1,3004	0,6427
117,2	0,965	0,035	1,6810	1,6762	0,8461	1,6762	0,8471
131,1	0,965	0,035	2,5810	2,5773	1,3471	2,5773	1,3511
144,4	0,965	0,035	3,7840	3,7731	2,0301	3,7731	2,0403
158,1	0,965	0,035	5,4440	5,4342	3,0004	5,4343	3,0236
168,5	0,965	0,035	7,0600	7,0475	3,9584	7,0476	3,9985
197,4	0,965	0,035	13,5900	13,5753	7,9154	13,5753	8,0623
223,7	0,965	0,035	22,9900	22,9396	13,6632	22,9399	14,0557
239,5	0,965	0,035	30,6400	30,5870	18,3438	30,5874	19,0072
255,3	0,965	0,035	40,0800	40,0602	24,0863	40,0609	25,1633
272,9	0,965	0,035	53,0700	53,0930	31,8644	53,0942	33,6382
98,7	0,944	0,056	0,8356	0,8358	0,4056	0,8358	0,4054
111,2	0,944	0,056	1,2900	1,2888	0,6508	1,2888	0,6512
123,9	0,944	0,056	1,9370	1,9378	1,0142	1,9378	1,0162
136,8	0,944	0,056	2,8460	2,8468	1,5384	2,8468	1,5444
151,9	0,944	0,056	4,3220	4,3175	2,4106	4,3175	2,4268
162,3	0,944	0,056	5,6420	5,6435	3,2136	5,6435	3,2427
188	0,944	0,056	10,3200	10,3244	6,1179	10,3246	6,2172
212,1	0,944	0,056	17,0600	17,0662	10,3816	17,0668	10,6424
236,4	0,944	0,056	26,8700	26,8999	16,6204	26,9017	17,2272
253,6	0,944	0,056	36,0400	36,1372	22,4315	36,1405	23,4697
78,3	0,916	0,084	0,3380	0,3375	0,1654	0,3375	0,1652
91,2	0,916	0,084	0,5614	0,5604	0,2891	0,5604	0,2888
104,3	0,916	0,084	0,9035	0,8999	0,4863	0,8999	0,4863
118	0,916	0,084	1,4230	1,4199	0,8010	1,4199	0,8021
131,4	0,916	0,084	2,1490	2,1429	1,2547	2,1429	1,2586
145,2	0,916	0,084	3,1850	3,1722	1,9211	3,1723	1,9318
159,2	0,916	0,084	4,5970	4,5872	2,8624	4,5874	2,8870
171,2	0,916	0,084	6,1880	6,1631	3,9330	6,1635	3,9790
197,2	0,916	0,084	11,0800	11,0348	7,3208	11,0368	7,4683
224,7	0,916	0,084	19,0800	19,0121	12,9539	19,0189	13,3695
249	0,916	0,084	29,2900	29,2162	20,1362	29,2335	21,0565
260,5	0,916	0,084	35,3000	35,2859	24,3655	35,3114	25,6639
273	0,916	0,084	42,9300	42,9054	29,6234	42,9431	31,4731
118,2	0,904	0,096	1,3260	1,3291	0,7994	1,3292	0,8006
145,9	0,904	0,096	3,0020	3,0053	1,9449	3,0054	1,9562
171,5	0,904	0,096	5,7530	5,7665	3,9319	5,7675	3,9788
199,2	0,904	0,096	10,6700	10,6871	7,5941	10,6917	7,7550

230,2	0,904	0,096	19,5000	19,5409	14,2739	19,5593	14,7797
82,6	0,861	0,139	0,2443	0,2433	0,1949	0,2432	0,1947
106,2	0,861	0,139	0,5962	0,5936	0,5094	0,5937	0,5095
130,5	0,861	0,139	1,3220	1,3157	1,1905	1,3167	1,1942
142,9	0,861	0,139	1,9080	1,8985	1,7535	1,9014	1,7629
154,8	0,861	0,139	2,6570	2,6413	2,4790	2,6484	2,4986
167,4	0,861	0,139	3,7000	3,6693	3,4900	3,6856	3,5289
82,6	0,849	0,151	0,2065	0,2082	0,1947	0,2082	0,1944
106,4	0,849	0,151	0,5259	0,5285	0,5126	0,5285	0,5127
130,4	0,849	0,151	1,1940	1,1968	1,1855	1,1994	1,1893
155,9	0,849	0,151	2,5750	2,5574	2,5549	2,5767	2,5760
167,6	0,849	0,151	3,5460	3,5066	3,5063	3,5464	3,5459
179,4	0,849	0,151	4,8000	4,7293	4,7289	4,8053	4,7994
111,5	0,834	0,166	0,6433	0,6446	0,6192	0,6453	0,6196
126,8	0,834	0,166	1,1200	1,1165	1,0538	1,1224	1,0566
152,4	0,834	0,166	2,5450	2,5165	2,3159	2,5591	2,3332
171,3	0,834	0,166	4,3330	4,2332	3,8621	4,3596	3,9103
181,4	0,834	0,166	5,6150	5,4445	4,9696	5,6513	5,0476
122	0,826	0,174	1,0890	1,0702	0,8973	1,0761	0,8991
135,1	0,826	0,174	1,7110	1,6760	1,3793	1,6958	1,3848
149,9	0,826	0,174	2,7350	2,6575	2,1573	2,7146	2,1722
166,5	0,826	0,174	4,3980	4,2157	3,4121	4,3649	3,4501
179,1	0,826	0,174	6,0790	5,7635	4,7027	6,0392	4,7733
77,8	0,806	0,194	0,2784	0,2799	0,1575	0,2781	0,1572
94,6	0,806	0,194	0,5983	0,5990	0,3243	0,5971	0,3240
106,1	0,806	0,194	0,9590	0,9592	0,5087	0,9598	0,5088
120,5	0,806	0,194	1,6490	1,6393	0,8552	1,6525	0,8567
132,6	0,806	0,194	2,4980	2,4608	1,2785	2,5016	1,2831
154,7	0,806	0,194	4,8370	4,6751	2,4835	4,8534	2,5037

Tableau 4 – Comparaison des pressions de rosée et de bulle d'un mélange (H_2O, HI), obtenues par H. Engels, eXMSL et Simulis Thermodynamics.

Le Tableau 4 compare les résultats produits respectivement par H. Engels, eXMSL et Simulis Thermodynamics. H. Engels recense uniquement des valeurs de pression de bulle, tandis que eXMSL et Simulis Thermodynamics ont été utilisés à la fois pour des calculs de pression de bulle et de pression de rosée. Tout comme les valeurs des coefficients d'activité apparents, les valeurs des pressions de bulle calculées par eXMSL et Simulis Thermodynamics sont quasiment identiques. Ces valeurs sont voisines des valeurs calculées par H.Engels, les différences étant liées, en partie ou en totalité, à l'expression différente de la pression de vapeur saturante de l'eau. Les valeurs de pression de rosée calculées par eXMSL et Simulis Thermodynamics sont par contre légèrement différentes, l'écart pouvant atteindre 0,5 pour cent. Afin de valider les résultats produits par chacun des deux environnements de simulation, nous testons la cohérence des calculs de pression de bulle et de température de bulle. Pour une composition apparente donnée $x_S^0 = 0{,}826$ $x_E^0 = 0{,}174$, et une température donnée $T_1 = 395{,}15K$, chaque environnement calcule la pression de bulle $P_b(T_1)$, puis la température de bulle à cette

pression, soit $T_2 = T_b(P_b(T_1))$. De même, pour une composition apparente donnée $x_S^0 = 0{,}826$ $x_E^0 = 0{,}174$, et une température donnée $T_1 = 395{,}15K$, chaque environnement calcule la pression de rosée $P_r(T_1)$, puis la température de rosée à cette pression, soit $T_3 = T_r(P_r(T_1))$. Le calcul est cohérent si et seulement si $T_1 = T_2 = T_3$. La Figure 62 présente un extrait de la session eXMSL correspondante. Le test de cohérence est parfaitement vérifié. Il en est de même pour toutes les valeurs des conditions initiales recensées dans (Engels 1990).

```
In[129]:= moleFractionSolvent = 0.825999999999999

Out[129]= 0.825999999999999

In[130]:= moleFractionElectrolyte = 0.173999999999999

Out[130]= 0.173999999999999

In[131]:= T = 395.149999999999

Out[131]= 395.149999999999

In[132]:= P = Apply[BubblePressure[H2O, HI], {T, moleFractionSolvent, moleFractionElectrolyte}]

Out[132]= 1.07020883937928

In[133]:= Apply[BubbleTemperature[H2O, HI], {P, moleFractionSolvent, moleFractionElectrolyte}]

Out[133]= 395.149999999941

In[134]:= P = Apply[DewPressure[H2O, HI], {T, moleFractionSolvent, moleFractionElectrolyte}]

Out[134]= 0.897314397000008

In[135]:= Apply[DewTemperature[H2O, HI], {P, moleFractionSolvent, moleFractionElectrolyte}]

Out[135]= 395.150000000239
```

Figure 62 – Test de cohérence des calculs de pression de bulle et de température de bulle par eXMSL.

Le Tableau 5 présente les résultats obtenus par Simulis Thermodynamics pour toutes les valeurs des conditions initiales recensées dans (Engels 1990). Le test de cohérence est moins bien vérifié : l'écart absolu maximal entre T_1 et T_2 vaut 0,8848K, l'écart absolu maximal entre T_1 et T_3 vaut 0,8573K, enfin l'écart absolu maximal entre T_2 et T_3 vaut 0,0444K.

Tableau 5 – Test de cohérence des calculs de pression de bulle et de température de bulle par Simulis Thermodynamics.

				Simulis Thermodynamics			
Conditions : Température T1 (°C)	Conditions : Pression (atm)	Mélange (Molaire) : WATER	Mélange (Molaire) : HYDROGEN IODIDE	Equilibres : Pression de bulle Pb(T1) (bar)	Equilibres : Température de bulle T2=Tb(Pb(T1)) (K)	Equilibres : Pression de rosée Pr(T1) (bar)	Equilibres : Température de rosée T3=Tr(Pr(T1)) (K)
107,1	1	0,983	0,017	1,2525	107,4853	0,6399	107,4547

132,4	1	0,983	0,017	2,7991	132,8462	1,5272	132,8140
157,3	1	0,983	0,017	5,5670	157,8106	3,1957	157,7769
170,7	1	0,983	0,017	7,7766	171,2468	4,5675	171,2126
181,9	1	0,983	0,017	10,1119	182,4780	6,0415	182,4434
206,4	1	0,983	0,017	17,1377	207,0482	10,5660	207,0142
230,7	1	0,983	0,017	27,3866	231,4201	17,2842	231,3898
254,9	1	0,983	0,017	41,7668	255,6931	26,7762	255,6701
267,2	1	0,983	0,017	51,0029	268,0305	32,8689	268,0123
90,7	1	0,965	0,035	0,6643	91,0481	0,3085	91,0177
95,7	1	0,965	0,035	0,8003	96,0592	0,3783	96,0283
109,5	1	0,965	0,035	1,3004	109,8908	0,6427	109,8587
117,2	1	0,965	0,035	1,6762	117,6091	0,8471	117,5762
131,1	1	0,965	0,035	2,5773	131,5432	1,3511	131,5091
144,4	1	0,965	0,035	3,7731	144,8772	2,0403	144,8418
158,1	1	0,965	0,035	5,4343	158,6134	3,0236	158,5768
168,5	1	0,965	0,035	7,0476	169,0416	3,9985	169,0043
197,4	1	0,965	0,035	13,5753	198,0232	8,0623	197,9851
223,7	1	0,965	0,035	22,9399	224,4005	14,0557	224,3651
239,5	1	0,965	0,035	30,5874	240,2481	19,0072	240,2162
255,3	1	0,965	0,035	40,0609	256,0966	25,1633	256,0695
272,9	1	0,965	0,035	53,0942	273,7518	33,6382	273,7302
98,7	1	0,944	0,056	0,8358	99,0661	0,4054	99,0341
111,2	1	0,944	0,056	1,2888	111,5952	0,6512	111,5617
123,9	1	0,944	0,056	1,9378	124,3261	1,0162	124,2909
136,8	1	0,944	0,056	2,8468	137,2585	1,5444	137,2219
151,9	1	0,944	0,056	4,3175	152,3980	2,4268	152,3596
162,3	1	0,944	0,056	5,6435	162,8260	3,2427	162,7865
188	1	0,944	0,056	10,3246	188,5978	6,2172	188,5566
212,1	1	0,944	0,056	17,0668	212,7681	10,6424	212,7276
236,4	1	0,944	0,056	26,9017	237,1416	17,2272	237,1049
253,6	1	0,944	0,056	36,1405	254,3955	23,4697	254,3628
280,9	1	0,944	0,056	55,5030	281,7848	36,6208	281,7573
78,3	1	0,916	0,084	0,3375	78,6214	0,1652	78,5913
91,2	1	0,916	0,084	0,5604	91,5497	0,2888	91,5176
104,3	1	0,916	0,084	0,8999	104,6798	0,4863	104,6456
118	1	0,916	0,084	1,4199	118,4127	0,8021	118,3764
131,4	1	0,916	0,084	2,1429	131,8461	1,2586	131,8079
145,2	1	0,916	0,084	3,1723	145,6819	1,9318	145,6417
159,2	1	0,916	0,084	4,5874	159,7193	2,8870	159,6773
171,2	1	0,916	0,084	6,1635	171,7523	3,9790	171,7091
197,2	1	0,916	0,084	11,0368	197,8264	7,4683	197,7820
224,7	1	0,916	0,084	19,0189	225,4082	13,3695	225,3662
249	1	0,916	0,084	29,2335	249,7842	21,0565	249,7466
260,5	1	0,916	0,084	35,3114	261,3216	25,6639	261,2862
273	1	0,916	0,084	42,9431	273,8635	31,4731	273,8296
118,2	1	0,904	0,096	1,3292	118,6135	0,8006	118,5767
145,9	1	0,904	0,096	3,0054	146,3841	1,9562	146,3431
171,5	1	0,904	0,096	5,7675	172,0534	3,9788	172,0095
199,2	1	0,904	0,096	10,6917	199,8320	7,7550	199,7875
230,2	1	0,904	0,096	19,5593	230,9241	14,7797	230,8838
256,2	1	0,904	0,096	30,6364	257,0060	23,6938	256,9713

271,3	1	0,904	0,096	38,9342	272,1560	30,4153	272,1237
82,6	1	0,861	0,139	0,2432	82,9236	0,1947	82,8994
106,2	1	0,861	0,139	0,5937	106,5751	0,5095	106,5491
130,5	1	0,861	0,139	1,3167	130,9304	1,1942	130,9048
142,9	1	0,861	0,139	1,9014	143,3594	1,7629	143,3349
154,8	1	0,861	0,139	2,6484	155,2875	2,4986	155,2648
167,4	1	0,861	0,139	3,6856	167,9179	3,5289	167,8977
82,6	1	0,849	0,151	0,2082	82,9132	0,1944	82,8993
106,4	1	0,849	0,151	0,5285	106,7613	0,5127	106,7494
130,4	1	0,849	0,151	1,1994	130,8119	1,1893	130,8044
155,9	1	0,849	0,151	2,5767	156,3690	2,5760	156,3674
167,6	1	0,849	0,151	3,5464	168,0971	3,5459	168,0980
179,4	1	0,849	0,151	4,8053	179,9275	4,7994	179,9300
111,5	1	0,834	0,166	0,6453	111,8488	0,6196	111,8606
126,8	1	0,834	0,166	1,1224	127,1806	1,0566	127,1957
152,4	1	0,834	0,166	2,5591	152,8416	2,3332	152,8583
171,3	1	0,834	0,166	4,3596	171,7966	3,9103	171,8077
181,4	1	0,834	0,166	5,6513	181,9306	5,0476	181,9354
122	1	0,826	0,174	1,0761	122,3646	0,8991	122,3844
135,1	1	0,826	0,174	1,6958	135,4955	1,3848	135,5154
149,9	1	0,826	0,174	2,7146	150,3355	2,1722	150,3519
166,5	1	0,826	0,174	4,3649	166,9889	3,4501	166,9948
179,1	1	0,826	0,174	6,0392	179,6365	4,7733	179,6289
77,8	1	0,806	0,194	0,2781	78,0738	0,1572	78,0895
94,6	1	0,806	0,194	0,5971	94,9069	0,3240	94,9237
106,1	1	0,806	0,194	0,9598	106,4322	0,5088	106,4484
120,5	1	0,806	0,194	1,6525	120,8679	0,8567	120,8807
132,6	1	0,806	0,194	2,5016	133,0028	1,2831	133,0091
154,7	1	0,806	0,194	4,8534	155,1832	2,5037	155,1638

La Figure 63 montre une meilleure cohérence des calculs de pression de rosée et de température de rosée proposés par eXMSL par comparaison avec Simulis Thermodynamics. Dans le cas de résultats produits par eXMSL les couples de valeurs T_1 et $T_3 = T_r(P_r(T_1))$ sont quasiment identiques, et le rapport T_3 / T_1 reste très voisin de 1. Dans le cas de résultats produits par Simulis Thermodynamics les couples de valeurs T_1 et $T_3 = T_r(P_r(T_1))$ diffèrent, T_3 étant toujours supérieur à T_1 de 0,3% à 0,4%.

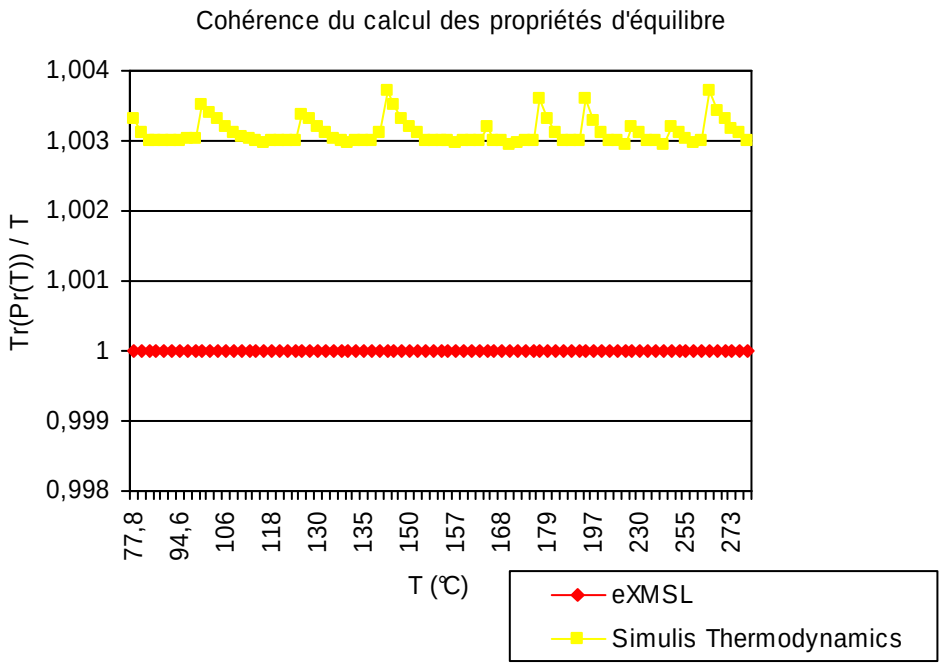

Figure 63 – Meilleure cohérence des calculs de pression de rosée et de température de rosée proposés par eXMSL par comparaison avec Simulis Thermodynamics.

L'étude présentée a validé la fiabilité de la résolution symbolico-numérique de certains modèles non linéaires indépendants du temps. La comparaison des valeurs obtenues par le système de calcul eXMSL avec des résultats publiés dans la littérature, ainsi qu'avec des résultats obtenus à l'aide d'un simulateur commercial apporte une première réponse positive à la question de l'intérêt de la démarche proposée consistant à mêler calcul formel et calcul numérique. La mise en œuvre de cette démarche sur un cas d'étude particulier suggère que la spécification des problèmes à l'aide du système de calcul symbolico-numérique proposé a plusieurs avantages. Comme cela a déjà été souligné, notamment en 1.1.4.1, la spécification du problème traité au sein d'un système de calcul formel a l'avantage connu d'être aisément évolutive car voisine de la spécification initiale sur papier. Ainsi, à partir de la spécification existante d'un binaire (H_2O, HI), il est facile d'étudier le comportement d'un autre binaire solvant-électrolyte pour lequel un autre modèle de coefficients d'activité est adopté. Outre ce type de réutilisation des spécifications de problèmes, que l'on peut qualifier de réutilisation par substitution (de règles de transformation), la démarche détaillée dans la partie 2.3, et complétée en 4.2.2.2, propose une réutilisation par composition et, de façon plus originale, une réutilisation par assemblage. Celle-ci est notamment rendue possible par la notion de fonction implicite proposée par le système de calcul eXMSL.

Le traitement d'un modèle de taille très modeste s'est accompagné cependant de temps de calcul très supérieurs aux temps de calcul obtenus à l'aide d'un simulateur numérique (Simulis Thermodynamics), anéantissant dans un premier temps les espoirs de performance placés dans

le modèle de coopération élaboré entre le calcul formel et le calcul numérique. Ce point négatif est particulièrement sensible lorsqu'une fonction implicite est définie à l'aide d'une, ou de plusieurs autres fonctions implicites, comme c'est le cas pour la température de rosée dans la modélisation adoptée. Enfin, la prise en compte des connaissances métier pour l'initialisation pertinente des variables a rappelé que le calcul formel ne saurait se substituer à une expertise dans un domaine.

4.2.4. Nomenclature :

S	Solvant
E	Électrolyte
C	Complexe
$P_S^0(T)$	Pression de vapeur saturante du solvant à la température T
$P_E^0(T)$	Pression de vapeur saturante de l'électrolyte à la température T
T	Température
x_S^0	Fraction molaire apparente du solvant
x_E^0	Fraction molaire apparente de l'électrolyte
x_S	Fraction molaire réelle du solvant en solution
x_E	Fraction molaire réelle de l'électrolyte en solution
x_C	Fraction molaire réelle de l'ensemble des complexes en solution
y_S	Fraction molaire du solvant en phase gazeuse
y_E	Fraction molaire de l'électrolyte en phase gazeuse
ν_S	Coefficient stœchiométrique du solvant dans l'équilibre de solvatation
ν_C	Coefficient stœchiométrique du complexe dans l'équilibre de solvatation

4.3. Distillation de Rayleigh

4.3.1. Présentation de la distillation batch

Depuis les années quatre-vingt, les procédés discontinus sont utilisés en production industrielle, dans le domaines de la chimie fine par exemple. Ils sont employés pour des fabrications de petit tonnage, pour lesquelles une mise en œuvre en continu demanderait des volumes trop petits. Ils sont également employés pour des opérations de fabrication délicates dans lesquelles interviennent soit des savoir-faire spécifiques, soit des opérations périodiques de nettoyage des appareils, soit des contraintes liées à la sécurité ou à la protection de l'environnement.

Au sein de ces procédés discontinus, la distillation discontinue occupe une place prédominante pour la séparation de produits à haute valeur ajoutée, pour lesquels les spécifications de pureté sont souvent contraignantes. De plus en plus cette distillation batch intervient également lors de la régénération de solvants, dans le but de diminuer les rejets polluants.

4.3.2. Modèle algébro-différentiel de la distillation de Rayleigh

Pour des mélanges à large spectre de volatilité, la distillation batch peut utiliser des colonnes sans alimentation. Le mélange à séparer est placé dans une cuve chauffée. Les constituants sont récupérés au distillat selon leur volatilité décroissante. La Figure 64 illustre ce procédé de distillation, discontinu puisque sans alimentation, et dynamique puisque les concentrations des constituants au bouilleur ou dans la colonne évoluent dans le temps.

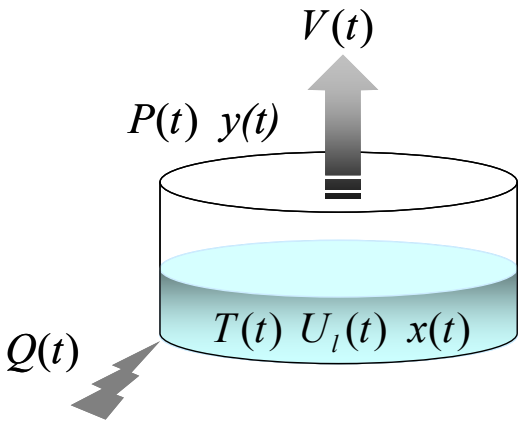

Figure 64 – Modèle de la distillation de Rayleigh.

Cette distillation sans reflux, dite de Rayleigh, peut être modélisée par un ensemble d'équations algébro-différentielles. En adoptant les notations recensées dans la nomenclature en

partie 4.3.7, Équation 26 exprime successivement un bilan matière global, un bilan matière partiel pour chaque constituant, un bilan enthalpique, et un équilibre entre la phase liquide et la phase gaz pour chaque constituant.

$$\begin{aligned}
&V(t) = -U_l'(t) \\
&\forall i \in \{1,\ldots,N\}; V(t)\cdot y_i(t) = -(U_l \cdot x_i)'(t) \\
&Q(t) - V(t)\cdot H(t) = (U_l \cdot h)'(t) \\
&\forall i \in \{1,\ldots,N\}; y_i(t) = K_i(t)\cdot x_i(t)
\end{aligned}$$

Équation 26 – Bilans et équilibres mis en jeu lors de la distillation de Rayleigh.

Équation 27 énumère les modèles particuliers utilisés pour évaluer respectivement l'enthalpie liquide du mélange, l'enthalpie vapeur du mélange et les constantes d'équilibre des différents constituants présents à la fois en phase liquide et en phase vapeur.

$$\begin{aligned}
&h(t) = mh(T(t), P(t), x(t)) \\
&H(t) = mH(T(t), P(t), y(t)) \\
&\forall i \in \{1,\ldots,N\}; K_i(t) = mK_i(T(t), P(t), x(t), y(t))
\end{aligned}$$

Équation 27 – Utilisation de modèles pour le calcul d'enthalpies et le calcul de constantes d'équilibre liquide-vapeur.

Équation 28 définit les fractions molaires des constituants de chaque phase en présence d'un gaz inerte, qui sera l'air la plupart du temps.

$$\sum_{i=1}^{N} x_i(t) = y_{inerte}(t) + \sum_{i=1}^{N} y_i(t)$$

Équation 28 – Définition des fractions molaires des constituants d'un système diphasique dans un environnement inerte.

Enfin, Équation 29 exprime le fait que le système se trouve dans un des deux états physiques :

- monophasique, pour lequel le débit vapeur $V(t)$ est nul ;
- diphasique, pour lequel la fraction molaire du gaz inerte $y_{inerte}(t)$ est nulle.

$$V(t) \cdot y_{inerte}(t) = 0$$

Équation 29 – Exclusion mutuelle des états mono-phasique et diphasique.

Le modèle algébro-différentiel de la distillation de Rayleigh s'obtient par concaténation de Équation 26, Équation 27, Équation 28 et Équation 29, ainsi que de conditions relatives aux variables dépendantes et, ou, à leurs dérivées par rapport à la variable dépendante, conditions exprimées à l'instant initial.

Le mélange étudié est le ternaire eau, méthanol et 1-butanol. La distillation de Rayleigh s'applique efficacement à ce mélange dont les constituants ont des températures d'ébullition relativement différentes. Un modèle thermodynamique à coefficients d'activité -NRTL- est utilisé pour la phase liquide fortement non idéale du fait des interactions entre l'eau et les alcools. Le domaine de pression utilisé justifie la modélisation de la phase vapeur comme un gaz parfait.

4.3.3. Calcul de conditions initiales cohérentes

Pour ce système dynamique particulier, l'étude consiste à choisir l'évolution temporelle de deux grandeurs[29], la puissance de chauffe au bouilleur et la pression, ainsi que les valeurs initiales de la température, de la rétention liquide et des fractions molaires liquide, dans le but de calculer l'évolution au cours du temps de toutes les variables dépendantes inconnues. D'un point de vue thermodynamique une telle initialisation est cohérente : elle est nécessaire et elle suffit pour déterminer toute grandeur présente dans le modèle à tout instant. D'un point de vue de la résolution du modèle par une méthode numérique, un calcul de conditions initiales cohérentes est requis préalablement à l'intégration numérique du système, comme indiqué en 2.2.1.3. Ce calcul détermine la valeur initiale de toutes les variables dépendantes et de leurs dérivées à l'ordre un.

Le dénombrement des variables dépendantes inconnues, des équations et des conditions initiales fournies justifie la stratégie exposée en 2.2.1.3 pour la détermination des conditions initiales cohérentes. Pour un ternaire, le nombre de variables dépendantes inconnues vaut 15. La recherche de conditions initiales cohérentes pour un intégrateur algébro-différentiel mettant en œuvre la méthode de Gear consiste à déterminer la valeur numérique de ces 15 variables

[29] Les grandeurs physiques du système dont l'évolution temporelle est connue a priori sont des variables dépendantes, mais ne sont pas des inconnues du système d'équations algébro-différentielles. Dans le cadre du modèle de système de calcul présenté, elles sont identifiées à des fonctions explicites.

inconnues en la valeur initiale de la variable indépendante, ainsi que la valeur numérique des dérivées de ces 15 variables inconnues en la valeur initiale de la variable indépendante. Le nombre d'équations différentielles vaut 5, le nombre d'équations algébriques vaut 10 et le nombre de conditions initiales fournies vaut 5. Par dérivation des 10 équations algébriques, 10 équations différentielles complémentaires sont obtenues. Le système non linéaire utilisé pour le calcul de conditions initiales cohérentes comprend donc :

- 15 équations non linéaires (par rapport aux variables dépendantes et à leurs dérivées évaluées en la valeur initiale de la variable indépendante), équations issues de l'évaluation de 5+10 équations différentielles en la valeur initiale de la variable indépendante ;
- 10 équations non linéaires (par rapport aux variables dépendantes évaluées en la valeur initiale de la variable indépendante), équations issues de l'évaluation des 10 équations algébriques en la valeur initiale de la variable indépendante.

Après substitution dans ce système non linéaire des 5 inconnues intervenant dans les conditions initiales par leurs valeurs numériques (température, rétention liquide et fractions molaires liquides), le système non linéaire comprend 25 équations entre 30-5=25 inconnues. Il s'agit d'un système d'équations non linéaires saturé pour lequel la formulation du schéma de Newton Équation 4 est appliquée. Le problème physique étant bien posé, le système d'équations non linéaires admet une solution.

Une première tentative de calcul symbolico-numérique par le système proposé de conditions initiales cohérentes échoue du fait de la singularité numérique[30] de la matrice Jacobienne associée au système d'équations non linéaires précédent. En effet, lorsqu'aucune estimation de la valeur numérique des inconnues d'un système d'équations non linéaires n'est fournie, par défaut le système de calcul initialise ces variables à zéro. Cette initialisation systématique, sans lien avec la physique du modèle, conduit à un mauvais conditionnement numérique de la matrice Jacobienne de la fonction résidu associée au calcul des conditions initiales cohérentes.

La nécessité de prendre en compte une estimation de la valeur initiale de certaines variables dépendantes lors de la résolution de systèmes d'équations algébro-différentielles impose d'enrichir la sémantique de la méthode **differentialAlgebraicEquationRoot** du système de

[30] La matrice Jacobienne de la fonction résidu associée au calcul des conditions initiales cohérentes n'est pas singulière. Toutefois, son mauvais conditionnement numérique la fait apparaître singulière aux yeux des routines de calcul numérique en charge de construire sa matrice inverse.

calcul proposé. Dans ce but, une nouvelle classe **ApproximatelyEqualExpression**, héritière de la classe **CompositeExpression**, complète la hiérarchie des classes modélisant les expressions mathématiques. La méthode **getValue** de cette classe retourne l'objet courant sans transformation. En effet, il s'agit uniquement de pouvoir disposer d'un moyen de construire des expressions mathématiques de la forme $a \approx b$. Ces expressions viendront, à côté des équations algébriques, des équations différentielles et des conditions initiales, donner une estimation numérique b de la valeur initiale a d'une variable dépendante donnée. Parallèlement à la définition de la classe **ApproximatelyEqualExpression**, une nouvelle méthode, **approximatelyEqual(a, b :Expression) : ApproximatelyEqualExpression**, vient compléter les services de la classe **ComputerAlgebraSystem**. Cette méthode est déclarée dans la classe **ComputerAlgebraSystem** et non dans la classe **CalculationSystem**. En effet, l'évaluation d'une expression **ApproximatelyEqualExpression** ne retournant jamais une quantité numérique, la création et l'utilisation de telles expressions a un sens au sein d'un système de calcul formel, mais n'en a pas au sein d'un système de calcul numérique.

Le nouveau service proposé par le système de calcul symbolico-numérique eXMSL permet de donner une estimation grossière des valeurs initiales des variables dépendantes absentes des conditions initiales. A titre d'exemple, les fractions molaires gazeuses des constituants présents dans les deux phases sont estimées par application d'équations d'équilibre de corps purs $\forall i \in \{1,...,N\}; y_i(0) = x_i(0) \cdot \frac{P_i^0(T(0))}{P(0)}$, la fraction molaire du gaz inerte étant alors estimée à partir de Équation 28. L'évaluation des modèles d'enthalpie aux points estimés produit par ailleurs une estimation des enthalpies molaires initiales.

Une seconde tentative de calcul symbolico-numérique par le système proposé de conditions initiales cohérentes échoue, encore une fois du fait de la singularité numérique de la matrice Jacobienne associée au système d'équations non linéaires. Après analyse, le mauvais conditionnement de cette matrice apparaît comme structurel. Les ordres de grandeur des fonctions coordonnées de la fonction résidu associée au calcul des conditions initiales cohérentes sont très différents : si la fonction coordonnée associée à Équation 28 prend ses valeurs autour de 1, les fonctions coordonnées associées aux modèles thermodynamiques prennent leurs valeurs dans des intervalles beaucoup plus grands[31]. Une mise à l'échelle (« scaling ») de certaines fonctions résidus, c'est-à-dire un pré conditionnement du système non linéaire à résoudre, s'impose pour produire des conditions initiales cohérentes. Un choix spécifique a été fait pour

[31] L'enthalpie molaire liquide d'un mélange vaut quelques dizaines de milliers de Joule en valeur absolue.

cette étude : l'équation initiale du modèle d'enthalpie molaire liquide du mélange est remplacée par Équation 30, de sorte que la fonction coordonnée correspondante dans la fonction résidu varie autour de l'unité. Cette mise à l'échelle suffit à supprimer la singularité numérique de la matrice Jacobienne et aboutit à un jeu de conditions initiales cohérentes.

$$\frac{h(t)}{10000} = \frac{mh(T(t), P(t), x(t))}{10000}$$

Équation 30 – Mise à l'échelle du modèle d'enthalpie molaire liquide.

Etant donnés un profil de pression et un profil de puissance de chauffe constants (Équation 31), ainsi que des conditions initiales imposées (Équation 32), la coopération du système de calcul eXMSL avec la bibliothèque mathématique IMSL conduit à des valeurs des conditions initiales cohérentes (Équation 33).

$$\forall t \in \Re^{+}; \begin{cases} P(t) = 1atm \\ Q(t) = 59378\mathrm{W} \end{cases}$$

Équation 31 – Choix des profils de pression et de température lors de la simulation de la distillation de Rayleigh.

$$\begin{aligned} &T(0) = 288{,}15K \\ &U_l(0) = 14242\mathrm{mol} \\ &\mathrm{x}_{\text{1-Butanol}}(0) = 0{,}185 \\ &\mathrm{x}_{\text{Méthanol}}(0) = 0{,}118 \\ &\mathrm{x}_{\text{Eau}}(0) = 0{,}697 \end{aligned}$$

Équation 32 – Conditions initiales imposées lors de la simulation de la distillation de Rayleigh.

$$
\begin{aligned}
&H_v(0) = -11{,}4564620964665\\
&h_l(0) = -45789{,}2426106912\\
&V(0) = 0{,}0000000000000000000015806352826980 7\\
&\mathrm{K}_{\text{1-Butanol}}(0) = 0{,}0101102461857102\\
&\mathrm{K}_{\text{Méthanol}}(0) = 0{,}0809098018943083\\
&\mathrm{K}_{\text{Eau}}(0) = 0{,}0000617842387315349\\
&\mathrm{y}_{\text{1-Butanol}}(0) = 0{,}00187039554435638\\
&\mathrm{y}_{\text{Méthanol}}(0) = 0{,}00954735662352838\\
&\mathrm{y}_{\text{Eau}}(0) = 0{,}0157848499676286\\
&\mathrm{y}_{\text{Air}}(0) = 0{,}972797397864486\\
&H_v'(0) = 0{,}0173659648657213\\
&T'(0) = 0{,}0424675451721244\\
&U_1'(0) = -0{,}0000000000000000000015806352826986 4\\
&V'(0) = 0{,}0000000000000000000013629513525029 2\\
&h_l'(0) = 4{,}16921780648785,\\
&\mathrm{K}_{\text{1-Butanol}}'(0) = 0{,}0000336091861227542\\
&\mathrm{K}_{\text{Méthanol}}'(0) = 0{,}000193263044053429\\
&\mathrm{K}_{\text{Eau}}'(0) = 0.0000617842387315349\\
&\mathrm{x}_{\text{1-Butanol}}'(0) = -0{,}0000000000000000000000000143692061072049\\
&\mathrm{x}_{\text{Méthanol}}'(0) = 0{,}0000000000000000000000000717411119146322\\
&\mathrm{x}_{\text{Eau}}'(0) = -0{,}0000000000000000000000000573719053592392\\
&\mathrm{y}_{\text{Air}}'(0) = -0{,}000072086353026894\\
&\mathrm{y}_{\text{1-Butanol}}'(0) = 0{,}00000621769943270953\\
&\mathrm{y}_{\text{Méthanol}}'(0) = 0{,}0000228050391983046\\
&\mathrm{y}_{\text{Eau}}'(0) = 0{,}0000430636143958798
\end{aligned}
$$

Équation 33 – Conditions initiales calculées lors de la simulation de la distillation de Rayleigh.

Bien qu'applicable à une classe limitée de systèmes d'équations algébro-différentielles, la méthode mise en œuvre ci-dessus montre tout l'intérêt d'un calcul automatique de conditions initiales cohérentes par le système de calcul symbolico-numérique : la spécification initiale du problème, c'est-à-dire le système d'équations algébro-différentielles, subit une séquence d'étapes de transformations formelles et d'étapes d'évaluations numériques, qui conduisent à un ensemble de conditions initiales cohérentes. Seule la mise à l'échelle de certaines équations, pour éviter la singularité numérique d'une matrice Jacobienne, est requise.

La classe de systèmes d'équations algébro-différentielles pour laquelle le système de calcul formel eXMSL détermine automatiquement des conditions initiales cohérentes peut être facilement étendue. La généralisation aux systèmes d'index de différentiation quelconque passe par la résolution d'un système non linéaire constitué par dérivation successive de la fonction résidu du système par rapport à la variable indépendante. (Campbell & Griepentrog 1995) désigne les équations de ce système comme étant les équations du vecteur dérivé (« derivative array equations »). (Campbell, Kelley, & Yeomans 1996) étudie l'application du schéma de Gauss-Newton lors de la résolution des équations du vecteur dérivé, en envisageant les différentes variantes de ce schéma. L'une d'entre elles, le schéma de Levenberg-Marquardt, prend en compte les singularités numériques de la matrice Jacobienne durant une itération. La coopération du système de calcul formel eXMSL, capable de produire automatiquement les équations du vecteur dérivé, et d'un système de calcul numérique codant le schéma de Levenberg-Marquardt, supprimerait toute intervention directe sur la spécification du problème lors du calcul de conditions initiales cohérentes.

4.3.4. Simulation de la distillation de Rayleigh dans l'état monophasique

La puissance de chauffe et la pression sont considérées comme constantes au cours du temps, égales aux valeurs indiquées dans Équation 31.

L'expression analytique des matrices Jacobiennes de la fonction résidu du système, par rapport aux variables dépendantes et par rapport aux dérivées des variables dépendantes, a été obtenue automatiquement par eXMSL à la suite du calcul des conditions initiales. Lors de l'intégration numérique, les matrices Jacobiennes sont évaluées numériquement en différents points par le système de calcul symbolico-numérique, à la demande du code mettant en œuvre la méthode de Gear. Les options par défaut de la fonction **D2SPG** de la bibliothèque mathématique IMSL ont été retenues. La spécification de la solution se limite donc à indiquer le nom de la routine numérique, sans aborder la question des tolérances par exemple.

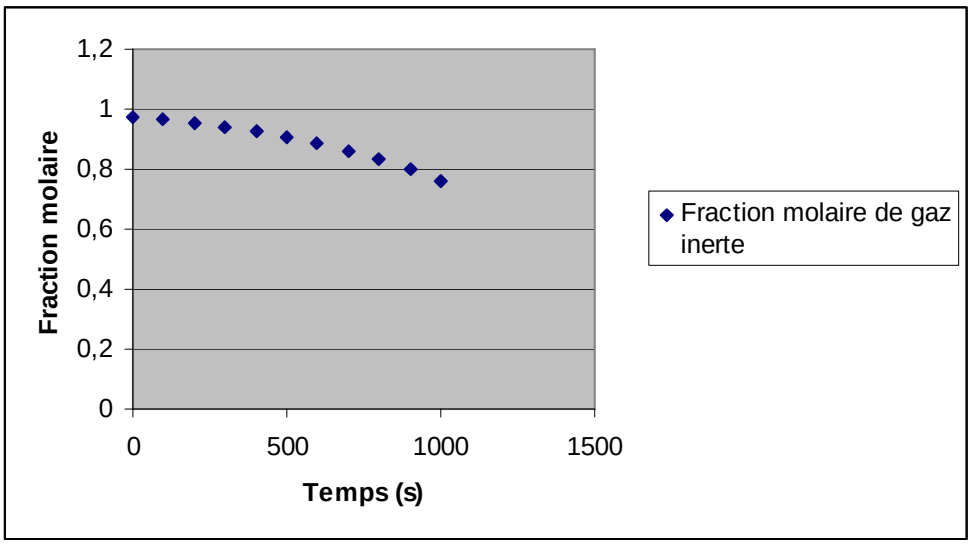

Figure 65 – Evolution de la fraction molaire de gaz inerte en cours de chauffe ($t \in [0,1000]$).

La qualité des conditions initiales cohérentes produites en 4.3.3 assure un démarrage correct du schéma prédicteur-correcteur. L'intégration numérique simule correctement le comportement physique du système dans l'état monophasique. Le débit de vapeur $V(t)$ demeure nul, la rétention liquide $U_l(t)$ et les fractions molaires liquide $x(t)$ demeurent constantes, égales à leurs valeurs initiales. La température du système $T(t)$ augmente du fait de l'apport extérieur d'énergie $Q(t)$. Enfin, la phase vapeur s'appauvrit en gaz inerte et s'enrichit en constituants présents dans la phase liquide. La Figure 65 illustre la diminution de la fraction molaire de gaz inerte au cours des 1000 premières secondes de chauffe.

4.3.5. Prise en compte du changement d'état du système thermodynamique

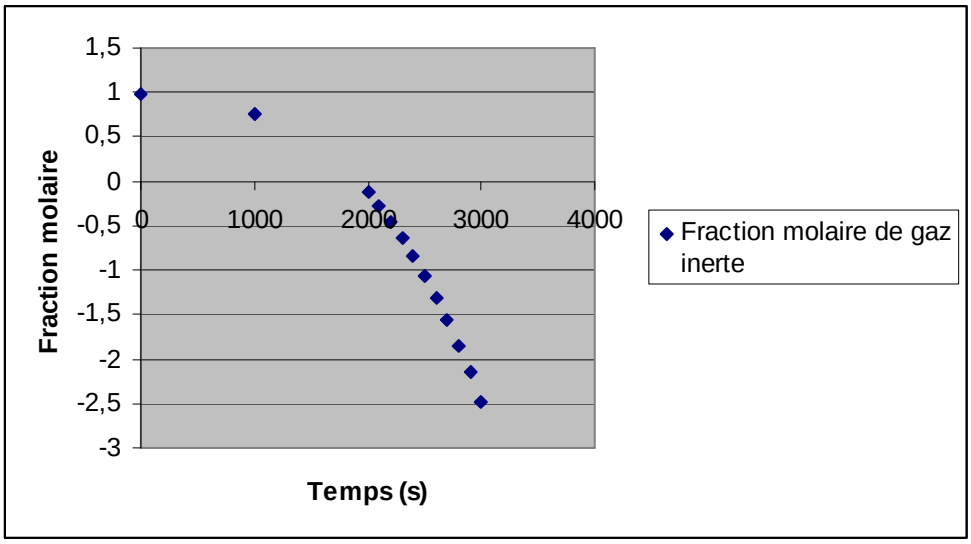

Figure 66 – Evolution de la fraction molaire de gaz inerte en cours de chauffe ($t \in [0,3000]$).

L'équation d'exclusion mutuelle des deux états possibles du système (Équation 29) est insuffisante pour que le système de calcul symbolico-numérique simule correctement le changement d'état du système thermodynamique : comme le montre la Figure 66, après avoir pris une valeur nulle, la fraction molaire du gaz inerte devient négative, et les fractions molaires des autres gaz augmentent jusqu'à devenir supérieures à un ! L'absence de contrainte sur le domaine de variation possible pour $y_{inerte}(t)$ explique ce comportement. Tant que $y_{inerte}(t) > 0$, la simulation rend bien compte du comportement du système dans l'état mono-phasique. Dès que $y_{inerte}(t) = 0$, la méthode d'intégration, de prédicteur-correcteur, décrit naturellement la solution continue et dérivable du modèle ci-dessus. Elle ignore la solution physique, qui elle est discontinue au point de bulle.

4.3.6. Simulation de la distillation de Rayleigh dans l'état diphasique

$$y_{inerte}(t) = b(t)^2$$

Équation 34 – Contrainte de positivité de la fraction molaire de gaz inerte.

La multiplicité des solutions mathématiques au modèle de distillation formulé ci-dessus, dont l'une ne représente pas le comportement du système physique dans l'état diphasique, constitue une limite voire une erreur de modélisation. L'ajout de la contrainte supplémentaire Équation 34, garantissant que $\forall t \in \Re^+ ; y_{inerte}(t) \geq 0$, conduit à un modèle dont l'unique solution représente le comportement du système physique dans l'état monophasique et dans l'état diphasique.

Hélas, si le modèle représente mieux la réalité physique, il impose à l'intégrateur numérique chargé de résoudre le système d'équations algébro-différentielles de traiter les événements d'état. Lors de cette étude, nous nous sommes volontairement limités à l'usage de l'intégrateur numérique livré avec la bibliothèque mathématique IMSL, qui ne gère pas les événements d'état ou de temps. La simulation de l'état diphasique a donc requis, suite à l'arrêt du système de calcul, lieu de la discontinuité, une nouvelle simulation à partir de l'état correspondant au point de bulle du système. Les valeurs des fractions molaires liquides et gazeuses, de la température, de la rétention liquide, des enthalpies molaires et des constantes d'équilibre ayant été calculées au point de discontinuité, le calcul de conditions initiales cohérentes a pour but la détermination du débit de vapeur et des grandeurs dérivées. L'initialisation du débit de vapeur à une valeur strictement positive suffit pour obtenir le jeu de conditions initiales cohérentes.

4.3.7. Nomenclature

Les symboles suivants désignent tous des fonctions de la variable indépendante t représentant le temps.

h	Enthalpie molaire du mélange en phase liquide
H	Enthalpie molaire du mélange en phase gazeuse
$K_i, i = 1,...,N$	Constantes d'équilibre liquide-gaz des différents constituants du mélange, hormis le gaz inerte
$P_i^0(T), i = 1,...,N$	Pressions de vapeur saturante des différents constituants du mélange, hormis le gaz inerte
Q	Puissance de la chauffe au bouilleur
mh	Modèle adopté pour exprimer l'enthalpie molaire du mélange en phase liquide
mH	Modèle adopté pour exprimer l'enthalpie molaire du mélange en phase gazeuse
$mK_i, i = 1,...,N$	Modèles adoptés pour exprimer les constantes d'équilibre liquide-gaz des différents constituants du mélange, hormis le gaz inerte
N	Nombre de constituants du mélange, hormis le gaz inerte
P	Pression
T	Température
U_l	Rétention liquide au bouilleur
V	Débit vapeur au-dessus de la phase liquide
x	Vecteur des fractions molaires des constituants du mélange en phase liquide
y	Vecteur des fractions molaires des constituants du mélange en phase gazeuse, hormis le gaz inerte
y_{inerte}	Fraction molaire du gaz inerte

4.4. Synthèse

Le Chapitre 4 analyse la simulation de quelques modèles continus, algébriques et différentiels, à partir d'un système appliquant le modèle de coopération entre calcul formel et calcul numérique énoncé au Chapitre 2.

Des conclusions positives relatives aux fonctionnalités du système de calcul produit peuvent être tirées :

1. il permet d'exprimer les modèles considérés dans une syntaxe proche d'un texte mathématique ;
2. il permet de spécifier les problèmes à résoudre dans une syntaxe proche de celle des appels à une bibliothèque mathématique ;
3. il calcule une solution fiable dans le cas des problèmes abordés ici.

Même si les temps de calcul des simulations sont pour l'instant très supérieurs à ceux obtenus par simulation numérique, l'obtention de tels résultats confirme tout l'intérêt d'une coopération entre calcul formel et calcul numérique.

Conclusion et Perspectives

L'objectif du travail de thèse présenté était de *« proposer un modèle original de coopération entre le calcul formel et le calcul numérique dans le but d'améliorer les processus de modélisation, de simulation et d'optimisation des systèmes statiques ou dynamiques »*.

Le 0 du manuscrit a analysé les deux principales approches adoptées habituellement pour faire collaborer le calcul formel et le calcul numérique. Celle présentée dans la partie 1.1, qui consiste à faire du calcul numérique à l'intérieur d'un système de calcul formel a l'avantage de mettre à disposition de la simulation à la fois les algorithmes de manipulation algébrique et bon nombre de méthodes de résolution numérique. La simulation peut alors intégrer des caractéristiques intéressantes, telles que le calcul à précision multiple ou les méthodes de résolution de l'analyse fonctionnelle. Pourtant les modèles complets et à caractère industriel, saisis et simulés dans un système de calcul formel demeurent rares, voire inexistants. Du fait de performances en temps de calcul moindres que les systèmes de calcul numérique, et d'une ouverture limitée vers ces derniers, les systèmes de calcul formel sont souvent envisagés à côté de la chaîne de modélisation et de simulation dans laquelle interviennent de multiples outils : mailleurs, bases de données, logiciels de visualisation, etc. Dans certains cas pourtant, comme le précise la partie 1.2, les systèmes de calcul formel sont intégrés en amont de cette chaîne de modélisation et de simulation pour synthétiser des codes de calcul après transformations formelles du modèle initial. Le code produit en langage de programmation standard, procédural ou à objet, bénéficie à la fois de la fiabilité des expressions mathématiques produites automatiquement par le calcul formel, et de la performance associée au calcul numérique lors de l'exécution du code compilé. Une utilisation des techniques du calcul formel tout au long du processus de simulation numérique, et pas seulement en amont, semble souhaitable. La partie 1.3 s'intéresse donc aux travaux où le calcul formel paraît être intégré à un environnement de calcul numérique. Ce n'est évidemment pas le cas lorsqu'une expression mathématique à la syntaxe d'un langage cible compilé est produite automatiquement par un système de calcul formel. Ce n'est toujours pas le cas lorsque les techniques de différentiation automatique sont appliquées pour produire un code évaluant une fonction et ses dérivées. Nous le rappelons pour conclure le 0 : seule la transformation d'expressions mathématiques en de nouvelles expressions mathématiques en cours d'exécution caractérise le calcul formel. Faire du calcul formel au sein d'un système de calcul numérique impose donc, non seulement de pouvoir évaluer

numériquement les expressions mathématiques, mais encore de pouvoir en produire de nouvelles par transformation formelle.

Forte de ce constat, le Chapitre 2 spécifie un système de calcul symbolico-numérique original où les expressions mathématiques subissent à la fois des étapes de transformation formelle et des étapes d'évaluation numérique. La conception de ce système, orientée par le modèle, c'est-à-dire les expressions mathématiques, établit le cadre général d'une coopération entre calcul formel et calcul numérique. L'évaluation symbolico-numérique des expressions est d'abord envisagée dans la partie 2.1, pour conduire à un modèle structurel général de système de calcul, qu'il soit formel ou numérique. Au vu de l'analyse des problèmes continus à résoudre, faite dans la partie 2.2, ce modèle est ensuite spécialisé pour spécifier le système de calcul symbolico-numérique proposé. Trois principes guident la conception :

1. l'utilisateur accède à des méthodes de résolution dont les interfaces sont celles proposées dans un système de calcul formel. La spécification du problème à résoudre se rapproche alors fortement du langage mathématique usuel, et n'intègre pas d'éléments empruntés à la spécification de la solution ;
2. la résolution utilise les méthodes d'un système de calcul numérique, méthodes efficaces et souvent éprouvées ;
3. le système de calcul symbolico-numérique se charge d'établir le lien entre le système de calcul numérique utilisé et les interfaces de calcul formel. Ceci consiste principalement à produire par transformations formelles les expressions mathématiques requises par la résolution numérique, et à évaluer numériquement à la demande, et en certains points, des expressions mathématiques symboliques.

La formalisation de la coopération entre calcul formel et calcul numérique se voit enrichie, dans la partie 2.3, de la notion de fonction implicite. Celle-ci, absente des systèmes de calcul formel, complète la notion de fonction explicite. L'introduction des fonctions implicites apporte un cadre fonctionnel plus riche pour la modélisation. Par ailleurs, la fiabilité du calcul des dérivées se voit systématiquement accrue par l'application d'un théorème des fonctions implicites. L'estimation du temps d'évaluation des dérivées des fonctions implicites par l'application de ce théorème s'avère inférieure à l'estimation du temps d'évaluation par une méthode aux différences finies lorsque la fonction implicite est définie par un système d'équations non linéaires. C'est l'inverse lorsque la fonction implicite est définie par un système d'équations algébro-différentielles. Dans le cas d'une fonction implicite définie par un problème d'optimisation une comparaison a priori est difficile.

La spécification du système de calcul symbolico-numérique proposé, initiée par la représentation des expressions mathématiques, puis complétée par la modélisation des services associés, a permis de distinguer clairement les spécificités des différents systèmes de calcul, formels ou numériques. Selon cette modélisation, un système de calcul symbolico-numérique est

une spécialisation d'un système de calcul formel. Les méthodes d'un ou de plusieurs systèmes de calcul numérique sont exploitées lors de la résolution de certaines classes de problèmes. Lors de l'exécution d'une méthode de résolution numérique, des expressions mathématiques produites par le système de calcul formel sont évaluées numériquement en certains points par le système de calcul numérique chargé de la résolution. Un système de calcul coopératif autour d'un modèle, les expressions mathématiques constituées de nombres et de symboles, apparaît clairement.

Le modèle général est complété dans la partie 2.4, par deux fonctionnalités importantes au vu des objectifs de fiabilité et de performance :

1. la réutilisation de codes numériques compilés existants ;
2. la réutilisation des expressions mathématiques créées précédemment dans le système.

Le complément 1 formalise l'encapsulation et l'appel à des fonctions boîtes noires mises à disposition du système de calcul symbolico-numérique, mais pour lesquelles le code source ne peut pas ou ne doit pas être accédé. Il s'agit d'enrichir le système de calcul coopératif afin que s'appliquent aux expressions mathématiques non seulement les méthodes de résolution numérique identifiées précédemment, mais aussi toute fonction pour laquelle les dérivées jusqu'à un certaine ordre peuvent être produites.

Le complément 2 vise un gain de performance en évitant la prolifération des expressions mathématiques créées par le système de calcul formel, et en réutilisant au mieux les résultats d'évaluations précédentes.

Les notions évoquées dans la partie 2.4 ne sont pas nouvelles, mais elles sont formalisées à l'aide d'un modèle de conception classique et de deux modèles de conception originaux. Elles conduisent finalement à un modèle de conception métier d'un système de calcul symbolico-numérique.

Le Chapitre 3 applique le modèle de conception élaboré précédemment pour produire un système de calcul symbolico-numérique particulier, eXMSL.

La partie 3.1 établit les règles de transformation du modèle de conception en modèle d'implémentation établi au sein de la « plate-forme » FORTRAN 90. Comme tout choix, ce choix de plate-forme présente à la fois des avantages et des inconvénients. L'avantage majeur est l'insertion dans une plate-forme particulièrement adaptée au calcul à hautes performances. Les principaux inconvénients de ce choix sont la non prise en compte du paradigme objet, et le nombre réduit des outils de génie logiciel amenés à supporter le cycle de vie du produit créé,

ainsi que leurs fonctionnalités limitées. L'établissement de règles claires et précises de transformation du modèle de conception vient pourtant garantir la génération systématique du modèle structurel d'implémentation. La génération automatique du modèle d'implémentation à partir du modèle de conception UML 2.0 pourra être envisagée à l'aide d'environnements de transformation de modèles sur la base des règles énoncées. Le modèle structurel du système de calcul symbolico-numérique se voit complété par un modèle comportemental, sous la forme de règles de transposition de l'algorithme des principales méthodes spécifiées dans le modèle de conception.

La partie 3.2 décrit les deux réalisations informatiques produites à partir du modèle d'implémentation de système de calcul symbolico-numérique. Une bibliothèque mathématique écrite en FORTRAN 90 vient renouveler le genre des bibliothèques mathématiques usuelles, en proposant à l'utilisateur une interface très proche de la spécification d'un problème sur le papier : les équations d'origine, mêlant nombres et symboles, sont fournies telles quelles aux méthodes de résolution qui, avant la résolution numérique proprement dite, génèrent automatiquement les expressions mathématiques formelles -fonctions résidus, matrices jacobiennes, dérivées directionnelles, ou conditions initiales cohérentes- dont l'évaluation est ou sera requise en cours de calcul. Un composant logiciel exploitant les standards « services Web », construit autour de cette bibliothèque mathématique, rend l'accès au système de calcul symbolico-numérique indépendant de la plate-forme FORTRAN 90, selon une architecture distribuée. Les expressions mathématiques sont alors représentées comme un flux MathML standardisé, donc susceptible d'être manipulé par différents outils, tels que des éditeurs d'équations ou d'autres systèmes de calcul formel.

L'utilisation du système eXMSL pour le traitement de deux problèmes particuliers, dans le Chapitre 4, vient confirmer ou infirmer les points forts de la démarche proposée.

Dans la partie 4.1, le calcul d'un modèle algébrique de solvatation, formulé de manière implicite, confirme l'intérêt et la fiabilité de la résolution numérique de systèmes non linéaires tirant avantage d'un calcul formel des résidus et de la dérivée de ces derniers. Ce résultat était attendu. En revanche, la faisabilité d'une résolution symbolico-numérique, et hiérarchique, d'un modèle composite se voit établie. Les modèles ou les sous-modèles sont associés à des fonctions, explicites, implicites, ou boîtes noires. Dès lors, résoudre un modèle revient à évaluer une fonction en un point, c'est-à-dire pour certaines valeurs des paramètres du modèle. Cette évaluation de la fonction modèle impose des évaluations des fonctions associées aux sous-modèles, et des évaluations de leurs dérivées lorsque la fonction modèle est implicite. Selon un principe identique, optimiser un modèle revient à évaluer la fonction objectif et sa dérivée pour

différentes valeurs des paramètres du modèle. Cette évaluation de la fonction modèle et de ses dérivées impose des évaluations des fonctions associées aux sous-modèles, et des évaluations de leurs dérivées. La résolution symbolico-numérique, et hiérarchique, du modèle de solvatation produit des résultats plus précis qu'une résolution hiérarchique au cours de laquelle les dérivées des sous-modèles sont obtenus par perturbation numérique. En effet, résoudre des modèles complexes par assemblage ou composition de modèles élémentaires, pour lesquels une estimation de la solution est connue, devient d'autant plus intéressant quand des dérivées précises des sous-modèles sont obtenues par application d'un théorème des fonctions implicites. Hélas, si l'objectif de fiabilité est atteint ici, la performance n'est pas au rendez-vous. Le temps de calcul d'une pression de rosée par eXMSL est bien plus important que le temps de calcul par un logiciel commercial Simulis Thermodynamics. Le surcoût est évidemment en partie lié à la gestion par le système de calcul formel de structures de données plus complexes que dans un environnement de calcul numérique. Une étude préalable d'analyse de performance nous encourage cependant à persévérer : une amélioration extrêmement importante des temps de calcul est à attendre de l'optimisation de deux mécanismes particuliers du système : la reconnaissance de motifs (*« pattern-matching »*), et l'application de fonctions à des arguments. L'amélioration sensible des performances du système de calcul proposé sera l'objectif premier des travaux à venir.

La partie 4.3 modélise et simule la distillation de Rayleigh, un processus physique décrit par un système d'équations algébro-différentielles. L'étude souligne tout l'intérêt de la production automatique par le système de calcul formel d'expressions mathématiques requises pour démarrer la simulation (conditions initiales cohérentes), pour la poursuivre (matrice jacobienne de l'opérateur algébro-différentiel), et enfin pour détecter des événements discrets. Les résultats obtenus confirment la fiabilité du système de calcul proposé. La robustesse du calcul permet de procéder à l'intégration numérique du modèle algébro-différentiel à partir de conditions initiales dont certaines sont estimées, et au calcul précis du point de bulle. Si les temps de calcul demeurent importants, la formulation explicite de ce problème limite cependant le surcoût lié au calcul formel. Le modèle traité représente de façon transparente deux domaines de comportement différent du système physique : le domaine mono-phasique et le domaine diphasique. Cette représentation continue par morceaux d'un système dynamique hybride est cependant rarement possible. Une première application du système de calcul proposé à la représentation des systèmes dynamiques hybrides pourrait prendre la forme d'un modèle dont tout ou partie serait une expression conditionnelle. Mais cette approche ne remplace évidemment pas un formalisme discret spécifique, plus approprié à l'étude de la sûreté de fonctionnement par exemple.

En résumé, le manuscrit propose un modèle de conception d'un système de calcul basé sur la coopération entre calcul formel et numérique. Il précise les transformations permettant de produire à partir de ce modèle de conception un modèle d'implémentation. Ce modèle conduit à une réalisation concrète, le système de calcul symbolico-numérique eXMSL. Au vu des résultats de deux études particulières, la démarche de calcul mixte proposée :

1. apporte plus d'expressivité lors de la modélisation. La spécification d'un problème prend la forme d'une fonction, le plus souvent implicite, modèle du système représenté. L'évaluation de cette fonction pour un jeu donné de paramètres donne lieu à une simulation particulière. La spécification du problème -la fonction modèle- est ici clairement distinguée de la spécification de la solution -les valeurs des paramètres du modèle, l'algorithme de résolution, le critère de convergence, …-. Les deux spécifications sont uniquement mises en correspondance lors de l'appel d'une méthode de résolution particulière ;
2. favorise la réutilisation de modèles par l'usage de la composition et de l'assemblage de fonctions ;
3. apporte plus de fiabilité lors de la simulation. Outre le calcul symbolique, le gain en fiabilité est lié à la modélisation des fonctions implicites dont les dérivées sont calculées par application d'un théorème des fonctions implicites.

En termes de perspectives de recherche, plusieurs pistes sont envisageables.

Le traitement des équations aux dérivées partielles, et la prise en compte des systèmes dynamiques hybrides pourraient compléter le système présenté.

L'amélioration des performances, la finalisation de l'accès aux composants numériques compatibles CAPE-OPEN, et l'intégration des composants eXMSL au sein d'environnements de résolution de problèmes existants (MATLAB, Microsoft Excel) constituent des préalables pour une diffusion du système proposé.

L'intérêt du système de calcul symbolico-numérique présenté ici, et plus généralement l'intérêt du modèle de coopération entre calcul formel et calcul numérique, sera avéré si l'objectif de performance est atteint. Cela suppose d'envisager un modèle de coopération adaptative entre calcul formel et calcul numérique. Les stratégies de coopération énoncées dans ce manuscrit demeurent génériques et prédéfinies. Si elles sont différentes suivant la classe mathématique du problème à résoudre -système d'équations non linéaires, système d'équations algébro-différentielles, ou minimisation d'un critère sous contraintes non linéaires- ces stratégies ne s'adaptent en rien à la nature précise de ce problème. Pour une classe donnée de problèmes, la séquence des étapes de transformation formelle et des étapes d'évaluation numérique est connue a priori. Pourtant, résoudre un système de deux équations non linéaires, ou résoudre un système

de deux millions d'équations non linéaires n'est en rien semblable : la fiabilité et la performance attendus lors de ces deux résolutions diffèrent, les méthodes numériques ou formelles diffèrent, et les moyens techniques à mettre en œuvre diffèrent. Si la spécification du problème est d'abord syntaxique, la spécification de la solution, telle que définie dans (Houstis & Rice 2000), est elle très sémantique. L'étude quantitative, menée dans les parties 2.3.1, 2.3.2 et 2.3.3, a esquissé les contours d'une démarche de choix entre calcul numérique et calcul formel pour résoudre un problème continu : le calcul de la dérivée d'une fonction implicite. Selon le seul critère de la performance, le choix est alors fonction de la nature mathématique des équations définissant la fonction implicite, du nombre de paramètres et du nombre de fonctions coordonnées. Plus généralement, le choix entre étapes de transformation formelle et étapes d'évaluation numérique lors d'une séquence de calculs est à envisager au vu de plusieurs critères, fiabilité et performance en premier lieu, eux-mêmes fonctions de multiples facteurs, tels que le conditionnement numérique, la taille du problème, l'architecture informatique utilisée, des paramètres utilisateurs, la sémantique du domaine métier considéré, …

La démarche de coopération entre calcul formel et calcul numérique doit donc s'adapter à la sémantique des modèles traités, et au contexte dans lequel a lieu la simulation, et ce en cours de simulation. Le cadre du modèle de coopération de type complémentaire, où *« l'objectif général* [du calcul] *n'est pas nécessairement partagé ni connu de tous les intervenants »*, demeure. L'affectation des tâches de calcul aux différents systèmes de calcul formel ou de calcul numérique se doit d'être évolutive. L'étude de scénarii de coopération de type complémentaire entre des codes de calcul existants et des codes synthétisés à l'aide de techniques de différentiation automatique (Alloula, Belaud, & Joulia 2000;Joulia, Alloula, & Belaud 1999) a déjà montré les limites de l'affectation statique des tâches de dérivation aux différents modules d'un simulateur de procédés. Le meilleur mode de calcul de la dérivée d'un modèle dépend de multiples paramètres (taille du modèle, nature explicite ou implicite, disponibilité de l'expression analytique, précision attendue, temps maximal de calcul exigé, etc.) dont la connaissance exhaustive n'est pas garantie a priori. Plus généralement, seule l'affectation évolutive des tâches de calcul aux différents sous-systèmes d'un environnement de résolution de problèmes, en fonction des compétences de chacun d'eux, semble garantir le meilleur compromis entre des objectifs souvent contradictoires. Dès lors, il ne s'agit plus seulement d'exhiber un modèle de coopération de type complémentaire entre calcul formel et calcul numérique, comme cela a été fait ici, mais de construire un modèle de coopération évolutive entre des systèmes de calcul. La nature précise, numérique ou formelle, du calcul proposé par chacun des systèmes élémentaires importe alors moins que leurs capacités à répondre à des objectifs fixés au niveau du système global. Un environnement de résolution de problèmes apparaît comme un système multi-agents,

« composé de plusieurs agents qui sont en relation, et mettent en œuvre des phénomènes collectifs » avec ici un objectif de résolution de problèmes particuliers dans des contextes particuliers. Ce modèle, où chaque système de calcul devient un agent, c'est-à-dire *« un programme autonome qui peut communiquer et coopérer avec d'autres agents »* ouvre un champ d'investigation pour accentuer le caractère coopératif du modèle présenté. Dans le domaine de la simulation des procédés, des travaux (Braunschweig 2002) ont déjà montré la pertinence de cette recherche.

Enfin, dans le cadre de l'ingénierie des procédés, une suite possible à ce travail est la définition d'un modèle de simulateur de procédés continus, acausal et réparti. La mise à disposition d'un formalisme fonctionnel enrichi, notamment par la notion de fonction implicite, peut conduire à une modélisation différente des systèmes continus. L'étude du modèle de solvatation d'Engels, faite dans la partie 4.2, a esquissé une stratégie de modélisation où les fonctions implicites, définies à partir des modèles, sont réutilisées par assemblage et composition. Il s'agit là en fait d'une formalisation du processus adopté en simulation modulaire continue : chaque module est construit à partir d'un modèle et d'une causalité particulière, les modules sont ensuite connectés entre eux, avant la simulation du procédé représenté. Ce que la simulation modulaire continue appelle « module » nous l'appelons « fonction », ce que la simulation modulaire continue appelle « connexion de modules », nous l'appelons « assemblage » ou « composition », selon la nature de la connexion : en parallèle ou en série. Dès lors, la simulation ou l'optimisation causales sont vues comme des processus d'évaluation et de dérivation de fonctions.

Aujourd'hui, les simulateurs de procédés continus orientés équations constituent le modèle d'un système par la mise à plat des équations caractérisant chacun de ses sous-systèmes, et ce avant la simulation. Contrairement aux simulateurs de procédés continus modulaires, *« l'implantation des modèles élémentaires [y] est indépendante d'une quelconque application particulière ou d'un quelconque algorithme qui pourra être utilisé pour leur résolution »* (de Pelegrini Soares & Secchi 2007). Ils sont donc particulièrement adaptés à la simulation acausale, puisque le modèle représentant le système simulé se compose d'équations où chaque symbole peut a priori jouer le rôle de paramètre ou de variable selon le problème posé. A l'inverse des simulateurs modulaires continus, où la causalité des modules est établie lors de leur définition, et

où la causalité du procédé simulé est établie dès la connexion des modules entre eux[32], le modèle de système de calcul proposé permet :

1. comme dans un simulateur continu orienté équations, la définition de modèles à partir d'expressions mathématiques ;
2. comme dans un langage orienté objets et équations, tel que Modelica, la connexion de modèles par l'identification de symboles appartenant à des modèles différents, ou par l'usage de règles de connexion ;
3. comme dans un simulateur continu orienté équations, la définition d'une causalité après avoir défini le modèle complet du procédé à simuler.

Au sein d'un simulateur modulaire continu, le modèle de système de calcul proposé permettrait de mener à bien la simulation ou l'optimisation d'un procédé complet à partir de l'évaluation automatique, et en cours de calcul, des modèles des différents sous-systèmes et de leurs dérivées. Cette modularité est intrinsèquement liée à l'usage du calcul formel, qui permet de différer la définition des modules, i.e. des fonctions caractéristiques de chaque sous-système, après la définition du modèle complet de procédé, et après le choix d'une causalité. Les simulateurs modulaires continus basés sur le seul calcul numérique ne peuvent avoir accès à une telle souplesse. En effet, ils ne peuvent transformer une expression mathématique -le modèle- en une autre expression mathématique -la fonction module- en cours d'exécution (cf. 1.4). Le modèle de coopération entre calcul formel et numérique pourrait être un moyen de réconcilier les adeptes de la simulation modulaire, soucieux de tirer parti de toute l'expertise métier lors de la simulation de chacun des modules, et les adeptes de la simulation orientée équations, désireux de maintenir une séparation entre la spécification des modèles et la spécification des solutions.

[32] Plus précisément, une causalité partielle est choisie lors de la conception d'un module. Le système d'équations caractéristique est sous-contraint de manière à laisser le choix d'un jeu de paramètres à l'utilisateur du simulateur.

Annexes

Annexe A. Réutilisation des codes numériques compilés existants

S'intégrer à des environnements de simulation numérique existants en limitant le périmètre modifié, comme cela a été déclaré en introduction du Chapitre 1, impose la réutilisation de codes numériques compilés existants. En effet, dans le cas de systèmes de modélisation et de simulation causaux (cf. MATLAB/Simulink), les instances de modèles sont toutes des fonctions écrites dans un langage de programmation procédurale, souvent compilé. Dans le cas de langages de modélisation orientés équations tels que Modelica, les instances de modèles incorporent tout de même des appels à des fonctions écrites dans un langage compilé, fonctions dont l'interface est déclarée dans le texte du modèle. Il est donc impératif dans le modèle de système de calcul proposé de pouvoir construire, puis évaluer, des expressions mathématiques qui mêlent à la fois des symboles, des nombres et des appels à des fonctions numériques compilées. La réutilisation de corrélations métier, de routines mettant en œuvre des heuristiques particulières très éloignées d'expressions analytiques, de fonctions accédant à des données tabulées ou extraites de bases de données, ou de bibliothèques mathématiques dont le code source n'est pas accessible impose de manipuler, au sein du même système de calcul, aussi bien des expressions mathématiques analytiques que des évaluations de fonctions compilées.

Etant donnée une fonction f de $\Re^n$ dans $\Re^m$, écrite dans un langage compilé, le système de résolution symbolico-numérique proposé doit permettre de créer et évaluer des expressions mathématiques formelles où apparaît un symbole f représentant la fonction numérique compilée. La cohérence de l'ensemble impose que le système d'évaluation symbolico-numérique des expressions satisfasse les trois règles suivantes :

1. l'évaluation du symbole f retourne f ;
2. si x_1, x_2, …, x_n sont des nombres, l'évaluation de l'expression $f(x_1, x_2, ..., x_n)$ produit le résultat numérique de l'appel à la fonction compilée f, converti dans le système choisi de typage des nombres. Si au moins une expression parmi x_1, x_2, …, x_n n'est pas un nombre, l'évaluation de $f(x_1, x_2, ..., x_n)$ produit $f(x_1, x_2, ..., x_n)$;
3. pour tout opérateur O défini de $F(\Re^n, \Re^m)$ dans $F(\Re^v, \Re^w)$, si x_1, x_2, …, x_n sont des nombres, l'évaluation de $\mathrm{O}(f)(x_1, x_2, ..., x_n)$ produit un résultat numérique compatible avec la sémantique de la fonction compilée f .

La mise en œuvre des règles 1 et 2 est évidente. La mise en œuvre de la règle 3 ne l'est pas. Parmi les opérateurs de $F(\Re^n, \Re^m)$ -ensemble des fonctions définies de $\Re^n$ dans $\Re^m$ - vers $F(\Re^v, \Re^w)$, nous avons choisi de nous limiter aux opérateurs de dérivation partielle. Dans ce

cas, il est possible de proposer un modèle d'évaluation satisfaisant cette règle. A partir du code compilé de la fonction f, et éventuellement des codes compilés de certaines dérivées partielles de cette fonction, est construite une routine calculant la valeur numérique d'une dérivée partielle de la fonction f, à un ordre quelconque $(\alpha_1, \alpha_2, ..., \alpha_n)$, en un point quelconque $(x_1, x_2, ..., x_n)$. Lorsque le code compilé du calcul de la dérivée partielle de f à un ordre $(\alpha_1, \alpha_2, ..., \alpha_n)$ n'est pas disponible, cette dérivée partielle peut être considérée comme identiquement nulle, peut être approchée par une formule aux différences finies basée sur le code de f, ou évaluée selon l'algorithme choisi par le concepteur du modèle. Si le code source de la fonction f est accessible, il est également possible de générer le code des dérivées d'ordre supérieur par différentiation automatique tel que présenté en 1.3.2. On voit dès lors que les techniques de calcul formel, de calcul numérique et de différentiation automatique peuvent collaborer à l'obtention d'une simulation fiable et performante.

Les considérations précédentes nous amènent à revoir le modèle structurel de fonctions tel qu'il a été d'abord exposé dans la partie 2.2.2, puis complété dans la partie 2.3.4. La définition d'une fonction est soit accessible au développeur du système, soit masquée. Le premier cas correspond aux fonctions dont l'expression analytique est fournie de façon explicite, et aux fonctions implicites évaluées en chaque point par résolution d'un système d'équations, ou par la minimisation d'un critère. Le second cas correspond aux fonctions dites « boîtes noires », dont l'évaluation est possible, mais dont tout ou partie du code source n'est pas disponible. La Figure 67 présente un diagramme de classes où sont représentées les différentes classes de fonctions. La notion de dérivation partielle y figure également avec la présence des classes concrètes **PartialDerivativeOperatorExpression** et **PartialDerivativeFunctionExpression** modélisant des expressions de la forme $\partial^{\text{orders}} f$ et $\partial^{\text{orders}} f(x)$ respectivement. La classe **CompositionExpression**, modélisant la composition de fonctions, complète l'éventail des familles de fonctions prises en compte par le système de calcul. Il est à noter que la hiérarchie de classes proposée ne tient absolument pas compte des propriétés mathématiques des fonctions, telle que la continuité ou la différentiabilité. L'unique critère de classification retenu est la manière dont les fonctions sont évaluées en un point : remplacement des paramètres formels par les arguments puis évaluation de l'expression mathématique obtenue, résolution d'un problème, appel à une fonction compilée, différentiation de la définition puis évaluation en un point ou règle de composition des fonctions. Les classes introduites dans le diagramme de la Figure 67 ont pour unique but d'organiser l'information structurelle relative à une fonction selon son type. Par conséquent les méthodes

getValue définies respectivement dans les classes **FunctionExpression**, **CompositionExpression**[33], **PartialDerivativeOperatorExpression** et **PartialDerivativeFunctionExpression** se limitent à retourner l'objet courant.

[33] La méthode **getValue** de la classe **CompositionExpression** peut retourner une expression simplifiée où toutes les fonctions identité ont été omises.

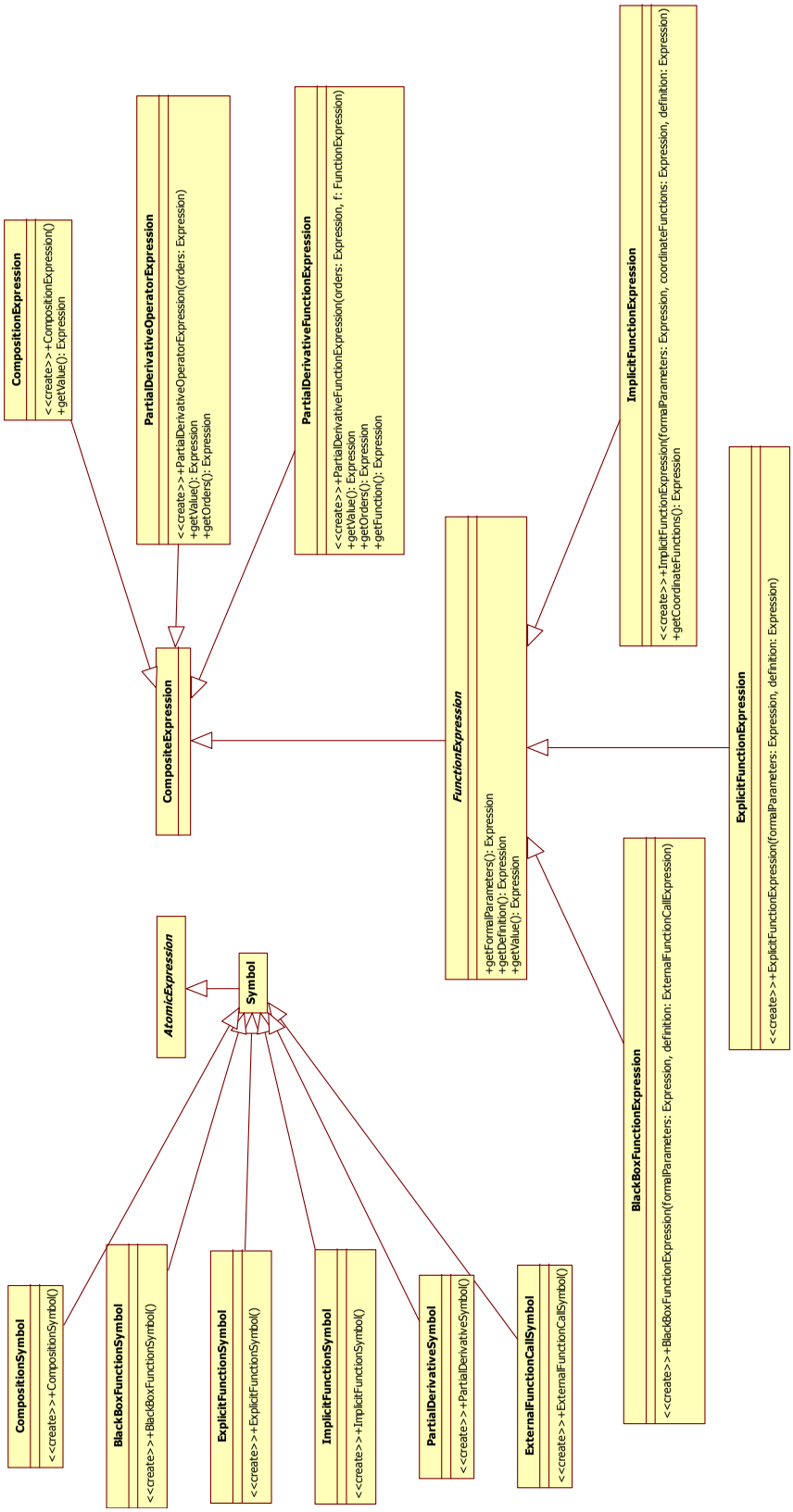

Figure 67 – Modèle structurel pour la définition de fonctions dans un système de calcul.

Les objets créés à partir de la classe **BlackBoxFunctionExpression** ne proposent pas l'accès à la définition complète de la fonction, car l'entité **BlackBoxFunctionExpression** modélise notamment les fonctions compilées et regroupées dans une bibliothèque, et les routines publiées dans

l'interface d'un composant logiciel. Toutefois, la signature de la fonction interfacée est accessible via une sous-expression de type **ExternalFunctionCallExpression**, type détaillé dans le diagramme de classes de la Figure 68.

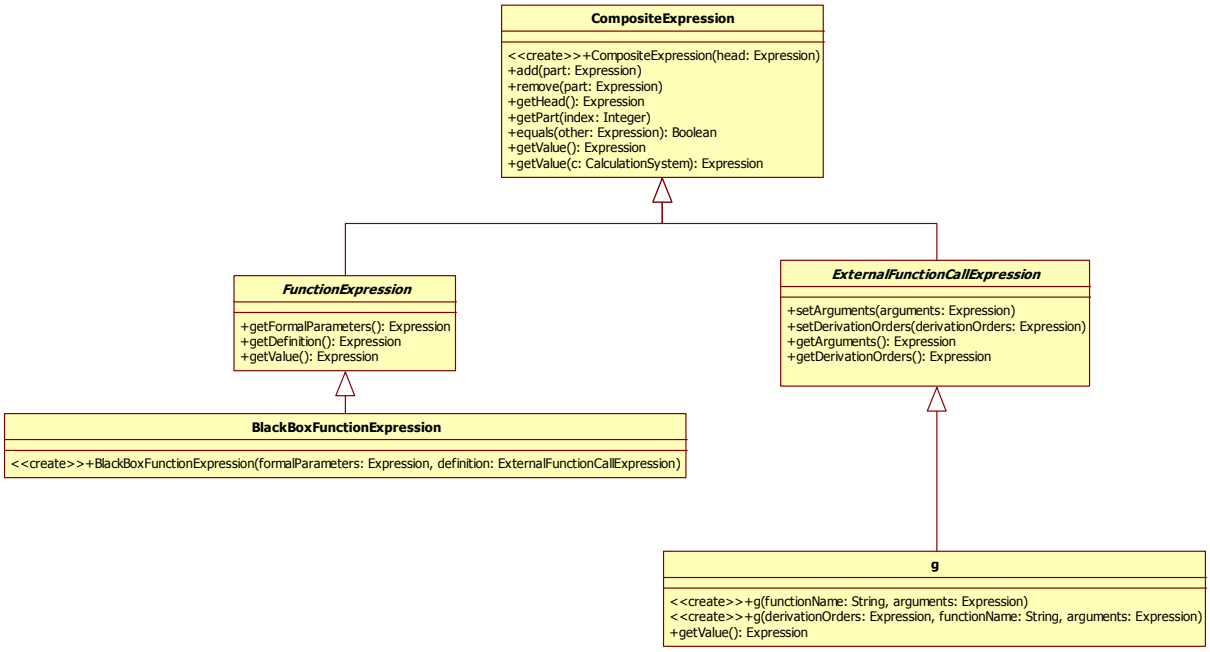

Figure 68 – Définition d'une fonction boîte noire dans le système de calcul proposé.

Les instances des sous-classes de la classe **ExternalFunctionCallExpression** représentent un appel à une fonction externe au système, ou à une de ses fonctions dérivées partielles, avec une liste particulière d'arguments. A titre d'illustration on s'intéresse à une fonction $g:\Re^3 \rightarrow \Re$ spécifiée en FORTRAN 90 par le bloc-interface suivant, mais dont la définition est inconnue :

```
INTERFACE
      REAL FUNCTION g(x, y, z)
            REAL, INTENT(IN) :: x, y, z
      END FUNCTION f
END INTERFACE
```

Afin de représenter, au sein d'une instance **c** du système de calcul symbolico-numérique, un appel *g(1.5,3,5)* à la fonction **g** avec la liste d'arguments *{1.5,3,5}*, une sous-classe **g** de **ExternalFunctionCallExpression** est définie. Cette sous-classe réalise la méthode **getValue** dont le rôle est d'appeler, avec les arguments préalablement transmis à l'objet par la méthode **setArguments**, la fonction **g** écrite dans le langage compilé. Si la classe **ExternalFunctionCallExpression** est proposée par le système de calcul symbolico-numérique, la définition de la classe **g** incombe à l'utilisateur du système. La méthode **getValue** doit appeler la fonction compilée avec la liste d'arguments fournie, après une conversion des types de données du système de calcul formel vers les types de données du système de calcul numérique, puis

retourner le résultat de l'appel après conversion de son type. Une fois la classe **g** disponible, une instance de cette classe est créée à l'aide d'une instruction[34] :

```
Expression gCall = new g(« g », c.list(c.real(1.5), c.integer(3),
c.integer(5))) ;
```

L'objet « gCall » est une expression composite ayant la structure arborescente suivante : le symbole prédéfini **ExternalFunctionCallExpression** comme tête, le symbole **g** comme première sous-expression et la liste ordonnée **{1.5,3,5}** comme seconde sous-expression. En adoptant la syntaxe du système Mathematica, cet objet peut s'écrire sous la forme : **ExternalFunctionCallExpression[g, {1.5,3,5}]**. Comme toute expression **gCall** peut être évalué à l'aide de l'instruction **gCall.getValue()**, qui retourne le résultat de l'appel **g(1.5,3,5)** à la fonction compilée.

Le modèle conceptuel de la Figure 68 autorise également la modélisation de l'évaluation d'une fonction dérivée partielle en un point, lorsque cette fonction est extérieure au système décrit. La manipulation de l'expression mathématique $\partial^{(2,0,1)}g(1.5,3,5)$ au sein du système de calcul symbolico-numérique impose à la méthode **getValue** de la classe **g** définie précédemment de pouvoir évaluer, non seulement la fonction externe *g* en tout point, mais également toutes ses fonctions dérivées partielles en tout point. Pour ce faire de multiples stratégies sont possibles, dont le choix est laissé au développeur de la classe **g** : la méthode **getValue** peut appeler des fonctions compilées existantes qui évaluent les dérivées partielles en un point, ou appliquer un schéma aux différences finies basé uniquement sur des appels à la fonction **g**. Dans le cas où le code source de la fonction **g** est accessible, une démarche de différentiation automatique est également possible pour obtenir le code de calcul des dérivées partielles jusqu'à un certain ordre. Enfin, dans le cas où la fonction **g** met principalement en jeu des expressions mathématiques algébriques, un système de calcul formel existant peut différentier ces expressions et synthétiser le code correspondant comme cela a été vu dans la partie 1.2.

Tout comme l'expression mathématique **g(1.5,3,5)**, l'expression mathématique $\partial^{(2,0,1)}g(1.5,3,5)$ est dès lors représentée par une instance de la classe **g** :

```
Expression gPartialDerivativeCall = new g(c.list(c. integer (2),
c.integer(0), c.integer(1)), « g », c.list(c.real(1.5), c.integer(3),
c.integer(5))) ;
```

[34] L'instruction adopte la syntaxe du langage Java.

En adoptant la syntaxe du système Mathematica, cet objet peut s'écrire sous la forme : **ExternalFunctionCallExpression[{2,0,1}, g, {1.5,3,5}]**. Comme toute expression, **gPartialDerivativeCall** peut être évalué à l'aide de l'instruction **gPartialDerivativeCall.getValue()** qui retourne le résultat de l'évaluation $\partial^{(2,0,1)} g(1.5,3,5)$ selon la stratégie choisie par le concepteur de la classe **g**.

Si les instances d'une sous-classe de **ExternalFunctionCallExpression** peuvent représenter des évaluations effectives d'une fonction externe, ou de ses dérivées partielles, en un point particulier, leur utilisation dans le cadre de la définition de fonctions boîtes noires est légèrement différente. Ces instances modélisent alors un appel générique à une fonction externe ou à une de ses dérivées. A titre d'exemple, la modélisation à partir de la fonction externe **g** introduite précédemment, d'une fonction **f** définie par : $f : \begin{array}{c} \Re^2 \to \Re \\ (u,v) \mapsto g(2+v,u.v,4) \end{array}$ passe par la création préalable d'un objet **ExternalFunctionCallExpression[g, {2+v, u v, 4}]**, où **u** et **v** sont des symboles. L'expression **BlackBoxFunction[{u, v}, ExternalFunctionCallExpression[g, {2+v, u v, 4}]]** définit alors une fonction boîte noire **f** par l'association à tout couple **{u, v}** de paramètres formels de la définition **ExternalFunctionCallExpression[g, {2+v, u v, 4}]**. Ici l'objet **ExternalFunctionCallExpression[g, {2+v, u v, 4}]** ne représente pas une évaluation effective de **g** au point **{2+v, u v, 4}**, évaluation qui n'a pas de sens dans un système de calcul numérique, mais bien une expression générique. Seule l'expression obtenue par substitution dans **ExternalFunctionCallExpression[g, {2+v, u v, 4}]** des arguments d'appel de **f** aux paramètres formels **{u, v}** sera évaluée, et uniquement dans le cas où les arguments d'appel sont des quantités numériques.

Pour résumer, une classe spécifique de chaque fonction externe doit être définie comme sous-classe de **ExternalFunctionCallExpression** afin d'établir le lien entre le système de calcul proposé et les ressources disponibles par ailleurs. Ceci étant fait, chaque fonction boîte noire est définie par la donnée de paramètres formels et d'une instance d'une sous-classe de **ExternalFunctionCallExpression**. Ce mode de définition des fonctions boîtes noires à partir d'un appel générique à une fonction externe offre une grande souplesse. Une fonction boîte noire peut modéliser exactement la fonction externe utilisée, comme dans l'expression **BlackBoxFunction[{u, v}, ExternalFunctionCallExpression[h, {u, v}]]**, ou modéliser la composée d'une fonction explicite avec la fonction externe utilisée, comme dans l'expression **BlackBoxFunction[{u, v}, ExternalFunctionCallExpression[g, {2+v, u v, 4}]]**. Cette dernière facilité permet l'adaptation des fonctions de calcul numérique disponibles à la représentation fonctionnelle souhaitée dans le système de calcul symbolico-numérique : un paramètre formel peut être fixé à une valeur constante, deux paramètres formels peuvent être permutés, ou être considérés comme identiques, un paramètre formel peut être vu comme une fonction de certains autres paramètres formels, etc.

Plus généralement encore, une fonction boîte noire peut modéliser la composée d'une fonction explicite avec une fonction dérivée partielle quelconque de la fonction externe utilisée, comme dans l'expression **BlackBoxFunction[{u, v}, ExternalFunctionCallExpression[{2,0,1}, g, {2+v, u v, 4}]]**. Cette facilité découle naturellement du fait que les instances des sous-classes de **ExternalFunctionCallExpression** peuvent non seulement représenter un appel à une fonction externe, mais également un appel à une fonction dérivée partielle d'une fonction externe.

A partir d'une fonction boîte noire désignée par **g**, l'utilisateur du système de calcul peut construire des expressions mathématiques mêlant des nombres, des symboles et des appels à la fonction **g** ou à ses dérivées partielles. Les expressions mathématiques symbolico-numériques peuvent comporter des dérivées directionnelles de la fonction **g** en un point donné et selon une direction donnée, ainsi que la matrice Jacobienne et la matrice Hessienne[35] de la fonction en un point donné, calculées à partir de la méthode **partialDerivative** de la classe **g**.

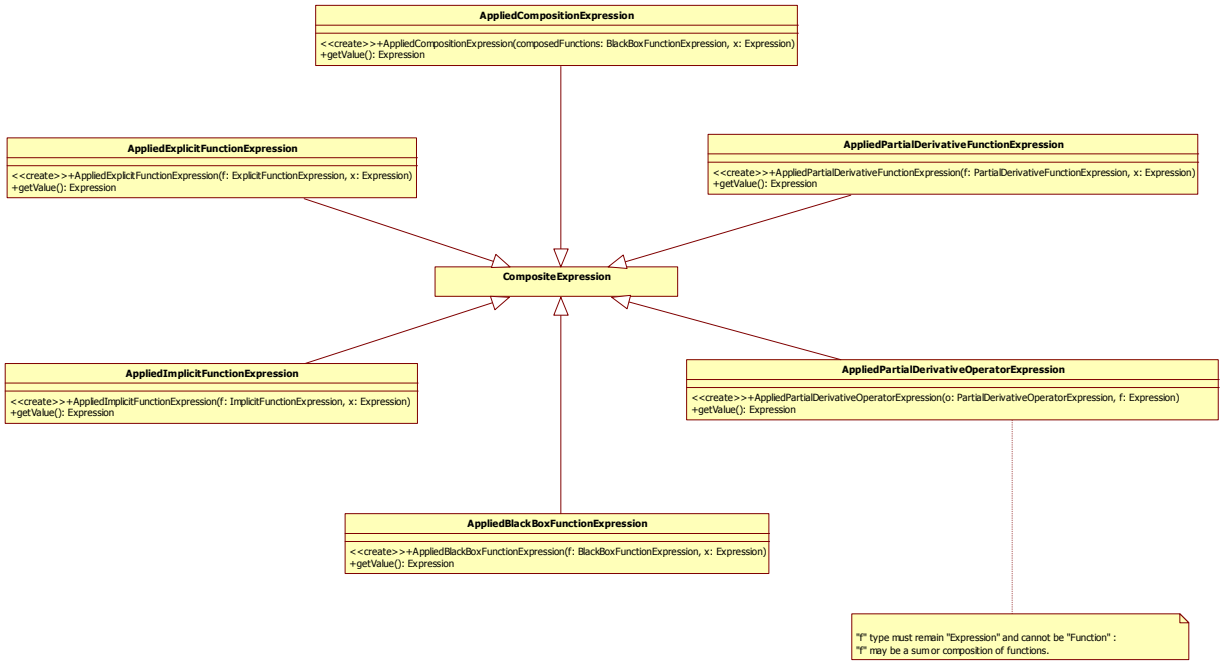

Figure 69 – Modèle structurel pour l'application de fonctions en un point dans un système de calcul.

La méthode **getValue** de la classe **FunctionExpression** est définie comme retournant systématiquement la fonction courante, que celle-ci soit explicite, implicite ou boîte noire. L'algorithme d'évaluation d'une fonction en un point, dépend par contre du type précis de la fonction. La nécessité de différencier les sémantiques de l'évaluation d'une fonction en un point selon la nature de la fonction, explicite, implicite ou boîte noire, se traduit naturellement dans le diagramme de classes de la Figure 69. Les définitions de la méthode **getValue** dans les classes

[35] La matrice Hessienne n'a évidemment de sens que dans le cas où la fonction g prend ses valeurs dans $\Re$.

AppliedExplicitFunctionExpression, **AppliedImplicitFunctionExpression** et **AppliedBlackBoxFunctionExpression** sont différentes. La sémantique retenue pour la méthode **getValue** de la classe **AppliedExplicitFunctionExpression** est celle adoptée par un système de calcul formel lors de l'évaluation d'une fonction explicite, telle qu'elle a été présentée en 2.2.2. La méthode **getValue** de la classe **AppliedImplicitFunctionExpression** résout le problème constituant la définition de la fonction implicite. Enfin, la méthode **getValue** de la classe **AppliedBlackBoxFunctionExpression** appelle la méthode **getValue** de l'objet **ExternalFunctionCallExpression** référencé par la fonction boîte noire[36].

La cohérence du modèle d'évaluation de fonctions en un point impose de prendre en compte trois cas particuliers, qui ne relèvent pas des sémantiques des méthodes **getValue** précédentes :

1. la fonction à appliquer est une composition de fonctions, et le point d'évaluation est un point de $\Re^n$. Ce cas correspond à la modélisation d'expressions mathématiques de la forme $(g \circ f)(x)$;
2. la fonction à appliquer est un opérateur de dérivation partielle, et le point d'évaluation est une fonction. Ce cas correspond à la modélisation d'expressions mathématiques de la forme $\partial^{\text{orders}} f$;
3. la fonction à appliquer est une fonction dérivée partielle, et le point d'évaluation est un point de $\Re^n$. Ce cas correspond à la modélisation d'expressions mathématiques de la forme $\partial^{\text{orders}} f(x)$.

L'introduction dans le diagramme de classes de la Figure 69 des définitions des classes **AppliedCompositionExpression**, **AppliedPartialDerivativeOperatorExpression** et **AppliedPartialDerivativeFunctionExpression** prend en charge respectivement les cas 1, 2 et 3.

Les tableaux qui suivent résument, selon le type de la fonction, les étapes d'évaluation d'une fonction en un point, de la dérivée partielle d'une fonction à un ordre donné, et de la dérivée partielle d'une fonction à un ordre et en un point donnés.

Etant donnés une instance **c** de la classe **CalculationSystem**, un objet *x* dont le type n'est pas un sous-type de **FunctionExpression**, et une instance **f** d'une sous-classe de **FunctionExpression**, le Tableau 6 et le Tableau 7 résument la manière dont est exécutée une instruction[37] **c.value(c.apply(f,x))**, construisant puis calculant l'expression mathématique *f(x)*. La généralité du mécanisme d'appel à des fonctions externes se traduit dans l'algorithme d' évaluation des

[36] Le Tableau 6 détaille l'appel à la fonction externe encapsulée par l'objet de type **BlackBoxFunctionExpression**.

[37] La syntaxe du langage Java est utilisée pour représenter les appels à une méthode à partir d'un objet.

fonctions boîtes noires. L'appel générique à la fonction externe est converti en un appel spécifique où interviennent les arguments d'appel à la fonction boîte noire. Afin de ne pas modifier la définition de la fonction initiale, l'appel générique est cloné puis modifié.

Tableau 6 – Etapes de construction et d'évaluation de l'expression $f(x)$ selon le type de la fonction f

Type de la fonction f	*ExplicitFunctionExpression*	*ImplicitFunctionExpression*	*BlackBoxFunctionExpression*
c.apply(f,x)	Retourne une *ApplyExpression*		
c.value(c.apply(f,x))	La méthode *getValue* de *ApplyExpression* construit une *AppliedExplicitFunctionExpression* La méthode *getValue* de *AppliedExplicitFunctionExpression* substitue les arguments d'appel *x* aux paramètres formels dans la définition de *f*, et évalue l'expression obtenue	La méthode *getValue* de *ApplyExpression* construit une *AppliedImplicitFunctionExpression* La méthode *getValue* de *AppliedImplicitFunctionExpression* substitue les arguments d'appel *x* aux paramètres formels dans la définition de *f*, et résout le problème obtenu	La méthode *getValue* de *ApplyExpression* construit une *AppliedBlackBoxFunctionExpression* La méthode *getValue* de *AppliedBlackBoxFunctionExpression* clone l'objet de type *ExternalFunctionCallExpression* référencé par l'objet de type *BlackBoxFunctionExpression*, appelle la méthode *getArguments* de l'objet cloné, substitue leurs valeurs numériques aux paramètres formels dans le résultat de *getArguments*, appelle la méthode *setArguments* de l'objet cloné avec le résultat de la substitution, et enfin appelle la méthode *getValue* de cet objet. Le résultat de cet appel à la méthode *getValue* est le résultat de l'évaluation.

Tableau 7 – Etapes de construction et d'évaluation de l'expression $f(x)$ selon le type de la fonction f

Type de la fonction f	*CompositionExpression*	*PartialDerivativeOperatorExpression*	*PartialDerivativeFunctionExpression*
c.apply(f,x)	Retourne une *ApplyExpression*		
c.value(c.apply(f,x))	La méthode *getValue* de *ApplyExpression* construit une *AppliedCompositionExpression*. La méthode *getValue* de *AppliedCompositionExpression* applique la définition de la composition de plusieurs fonctions pour produire le résultat de l'évaluation.	La méthode *getValue* de *ApplyExpression* construit une *AppliedPartialDerivativeOperatorExpression*. La méthode *getValue* de *AppliedPartialDerivativeOperatorExpression* est exécutée : *x* n'étant pas une fonction, cette évaluation déclenche une exception.	La méthode *getValue* de *ApplyExpression* construit une *AppliedPartialDerivativeFunctionExpression*. Si *f.getFunction()* est une *ImplicitFunctionExpression*, *x* est une quantité numérique et |*orders*|=1, alors la méthode *getValue* de *AppliedPartialDerivativeFunctionExpression* exécute *c.value(c.jacobianMatrix(f, x))* et extrait de la matrice Jacobienne la dérivée partielle souhaitée qui est le résultat de l'évaluation. Si une

			des conditions précédentes n'est pas vérifiée, la méthode *getValue* de *AppliedPartialDerivativeFunction Expression* retourne l'objet courant qui est le résultat de l'évaluation.

Etant données une instance *c* de la classe **CalculationSystem** et une instance **f** d'une sous-classe de **FunctionExpression**, le Tableau 8 et le Tableau 9 résument la manière dont est exécutée l'instruction **c.value(c.apply(c.partialDerivative(orders), f))** construisant puis calculant l'expression mathématique $\partial^{\text{orders}} f$. Une fonction dérivée partielle d'une fonction boîte noire est construite simplement en modifiant les ordres de dérivation partielle dans l'appel générique à la fonction externe. Afin de ne pas modifier la définition de la fonction initiale, l'appel générique est cloné puis modifié. La fonction dérivée partielle d'une fonction composée est vue comme une fonction explicite dont la définition est obtenue par calcul formel en appliquant successivement, lors de chaque dérivation partielle par rapport à une variable, le théorème de dérivation d'une fonction composée. La méthode **getValue** de la classe **AppliedPartialDerivativeOperatorExpression** traduit les règles de transformation suivantes : $\partial^{\alpha}\partial^{\beta} = \partial^{\alpha+\beta}$ et $\partial^{\alpha}(\partial^{\beta} f) = \partial^{\alpha+\beta} f$, respectivement utilisées lors de l'évaluation de la dérivée partielle d'un objet de type **PartialDerivativeOperatorExpression** et d'un objet de type **PartialDerivativeFunctionExpression**.

Tableau 8 – Etapes de construction et d'évaluation de l'expression $\partial^{\text{orders}} f$ selon la nature de la fonction f

Type de la fonction f	*ExplicitFunctionExpression*	*ImplicitFunctionExpression*	*BlackBoxFunctionExpression*
c.apply(c.partialDerivative(orders), f)	Retourne une *ApplyExpression*		
c.value(c.apply(c.partialDerivative(orders), f))	La méthode *getValue* de *ApplyExpression* construit une *AppliedPartialDerivativeOperatorExpression*.		

	La méthode *getValue* de *AppliedPartialDerivativeOperatorExpression* construit une *ExplicitFunctionExpression* dont la définition est obtenue par application du théorème de dérivation d'une fonction composée. La fonction explicite ainsi construite est le résultat de l'évaluation.	La méthode *getValue* de *AppliedPartialDerivativeOperatorExpression* construit une *PartialDerivativeFunctionExpression* par appel du constructeur avec les arguments *orders* et *f*.	La méthode *getValue* de *AppliedPartialDerivativeOperatorExpression* clone l'objet de type *ExternalFunctionCallExpression* référencé par l'objet de type *BlackBoxFunctionExpression*, appelle la méthode *setDerivationOrders* de l'objet cloné *w* avec pour argument *c.plus(w.getDerivationOrders(), orders)*, et enfin construit un nouvel objet de type *BlackBoxFunctionExpression* à partir de *f.getFormalParameters()* et de l'objet *w*. La fonction boîte noire ainsi construite est le résultat de l'évaluation.

Tableau 9 – Etapes de construction et d'évaluation de l'expression $\partial^{\text{orders}} f$ selon la nature de la fonction f

Type de la fonction f	*CompositionExpression*	*PartialDerivativeOperatorExpression*	*PartialDerivativeFunctionExpression*
c.apply(c.partialDerivative(orders), f)	Retourne une expression du type *ApplyExpression*		
c.value(c.apply(c.partialDerivative(orders), f))	La méthode *getValue* de *ApplyExpression* construit une *AppliedPartialDerivativeOperatorExpression*.		
	La méthode *getValue* de *AppliedPartialDerivativeOperatorExpression* construit une *ExplicitFunctionExpression* dont la définition est obtenue par application de la règle de dérivation des fonctions composée. L'expression ainsi construite est le résultat de l'évaluation.	La méthode *getValue* de *AppliedPartialDerivativeOperatorExpression* construit une *PartialDerivativeOperatorExpression* par appel du constructeur avec l'argument *c.plus(f.getDerivationOrders(), orders)*. L'expression ainsi construite est le résultat de l'évaluation.	La méthode *getValue* de *AppliedPartialDerivativeOperatorExpression* construit une *PartialDerivativeFunctionExpression* par appel du constructeur avec les arguments *c.plus(f.getDerivationOrders(), orders)* et *f.getFunction()*. L'expression ainsi construite est le résultat de l'évaluation.

Etant donnés une instance **c** de la classe **CalculationSystem**, un objet **x** dont le type n'est pas un sous-type de **FunctionExpression**, un objet **orders** de même longueur que **x** si **x** est une liste, et une instance **f** d'une sous-classe de **FunctionExpression**, le Tableau 10 résume la manière dont est exécutée une instruction[38] **c.value(c.apply(c.apply(c.partialDerivative(orders), f), x))**, construisant

[38] La syntaxe du langage Java est utilisée pour représenter les appels à une méthode à partir d'un objet.

puis calculant l'expression mathématique $\partial^{\text{orders}} f(x)$. Si **f** est une fonction implicite[39], et l'argument d'appel une quantité numérique, l'expression est calculée par application d'un théorème des fonctions implicites. Les dérivées partielles de longueur supérieure à un d'une fonction implicite pourraient être traitées par application d'un théorème des fonctions implicites, selon le principe exposé dans la partie. Pour l'instant, le système de calcul symbolico-numérique proposé restreint les méthodes numériques de résolution aux méthodes d'ordre un lorsque les modèles sont exprimés à l'aide de fonctions implicites . Si f n'est pas une fonction implicite, l'expression $\partial^{\text{orders}} f(x)$ reste inchangée. Dans ce cas **f** n'est ni une fonction explicite dont l'évaluation de la dérivée partielle est une autre fonction explicite, ni une fonction boîte noire dont l'évaluation de la dérivée partielle est une autre fonction boîte noire, ni une fonction composée dont l'évaluation de la dérivée partielle est une fonction explicite, ni un opérateur de dérivation partielle ∂^{β} car dans ce cas l'expression $\partial^{orders}\partial^{\beta}$ aurait été évaluée en $\partial^{orders+\beta}$, ni une fonction dérivée partielle $\partial^{\beta} f$ car dans ce cas l'expression $\partial^{orders}(\partial^{\beta} f)$ aurait été évaluée en $\partial^{orders+\beta} f$.

Tableau 10 – Etapes de construction et d'évaluation de l'expression $\partial^{\text{orders}} f(x)$ selon la nature de la fonction f

Type de la fonction f	*ImplicitFunctionExpression*	Autre
c.apply(c.apply(c.partialDerivative(orders), f), x)	Retourne une *ApplyExpression*	
c.value(c.apply(c.apply(c.partialDerivative(orders), f), x))	La méthode *value* de *SymbolicNumericalCalculationSystem* évalue la sous-expression *c.apply(c.partialDerivative(orders), f)*.	

[39] Pour l'instant, seules les dérivées partielles de longueur un sont calculées par application d'un théorème des fonctions implicites, ainsi que les dérivées de longueur deux des fonctions à valeurs dans $\Re$. Ce dernier calcule permet l'évaluation précise de matrices hessiennes pour l'optimisation.

	D'après le Tableau 8, le résultat de cette évaluation est une *PartialDerivativeFunctionExpression*. Si *x* est une quantité numérique et |*orders*|=1, alors la méthode *getValue* de *AppliedPartialDerivativeFunctionExpression* exécute *c.value(c.jacobianMatrix(f, x))* et extrait de la matrice Jacobienne la dérivée partielle souhaitée qui est le résultat de l'évaluation. Si une des conditions précédentes n'est pas vérifiée, la méthode *getValue* de *AppliedPartialDerivativeFunctionExpression* retourne l'objet courant qui est le résultat de l'évaluation.	D'après le Tableau 8 et le Tableau 9, le résultat de cette évaluation est du type *ExplicitFunctionExpression*, *BlackBoxFunctionExpression*, *PartialDerivativeOperatorExpression* ou *PartialDerivativeFunctionExpression*. Le Tableau 6 et le Tableau 7 détaillent l'évaluation de l'application d'une fonction en un point.

Dans le cadre du travail présenté, les opérateurs de composition et de dérivation partielle appliqués à une fonction boîte noire sont traités de manière cohérente. Une fonction boîte noire peut être composée avec tout autre type de fonction, sous réserve que la composition soit valide du point de vue des dimensions des espaces d'arrivée et de départ. Par ailleurs la dérivation partielle d'une fonction boîte noire, définie de $\Re^n$ dans $\Re^m$, puis l'évaluation de la fonction résultat en un point de $\Re^n$ retourne une valeur numérique.

Dans le cas de fonctions explicites ou boîtes noires, les opérateurs **jacobianMatrix**, **hessianMatrix** et **directionalDerivative**, i.e. les opérateurs de dérivation d'ordre un et deux, et la dérivation directionnelle, sont construits à partir de l'opérateur de dérivation partielle **partialDerivative**. Dans le cas de fonctions implicites c'est l'inverse : les opérateurs **jacobianMatrix**, **hessianMatrix** et **directionalDerivative**, i.e. les opérateurs de dérivation d'ordre un et deux, et la dérivation directionnelle, sont construits à partir des équations obtenues par application d'un théorème des fonctions implicites ; l'opérateur de dérivation partielle **partialDerivative** est construit à partir de l'opérateur **jacobianMatrix** lorsque la longueur de l'ordre de dérivation est égale à un. Dans le cas de fonctions implicites, et lorsque la longueur de l'ordre de dérivation est égale à deux ou plus, **partialDerivative** ne procède pas au calcul des dérivées partielles en appliquant un théorème des fonctions implicites. En effet, ce calcul nécessiterait la détermination successive de toutes les dérivées partielles de longueur intermédiaire. Si elles sont nécessaires, ces dérivées peuvent être approchées par un schéma aux différences finies.

```
SymbolicNumericalCalculationSystem
<<create>>+SymbolicNumericalCalculationSystem(numericalCalculationSystem: NumericalCalculationSystem)
+integer(n: Integer): Integer
+rational(n: Integer, d: Integer): Rational
+real(r: Float): Real
+complex(r: Float, i: Float): Complex
+symbol(name: String): Symbol
+plus(a: Expression, b: Expression): PlusExpression
+times(a: Expression, b: Expression): TimesExpression
+power(a: Expression, b: Expression): PowerExpression
+cos(a: Expression): CosExpression
+set(leftHandSide: Expression, rightHandSide: Expression, : SetExpression)
+expand(a: Expression): ExpandExpression
+factor(a: Expression): FactorExpression
+simplify(a: Expression): SimplifyExpression
+list(): ListExpression
+list(a: Expression): ListExpression
+list(a: Expression, b: Expression): ListExpression
+apply(f: Expression, list: Expression): ApplyExpression
+function(formalParameters: Expression, definition: Expression): ExplicitFunctionExpression
+function(formalParameters: Expression, coordinateFunctions: Expression, definition: Expression): ImplicitFunctionExpression
+function(formalParameters: Expression, externalFunctionCall: ExternalFunctionCallExpression): BlackBoxFunctionExpression
+compose(f: Expression, g: Expression): CompositionExpression
+partialDerivative(orders: Expression): PartialDerivativeOperatorExpression
+jacobianMatrix(f: Expression, x: Expression): JacobianMatrixExpression
+hessianMatrix(f: Expression, x: Expression): HessianMatrixExpression
+directionalDerivative(f: Expression, x: Expression, direction: Expression): DirectionalDerivativeExpression
+linearEquationRoot(equations: Expression, variables: Expression, initialGuess: Expression): LinearEquationRootExpression
+nonLinearEquationRoot(equations: Expression, variables: Expression, initialGuess: Expression): NonLinearEquationRootExpression
+nonLinearEquationRoot(residualEvaluation: FunctionExpression, residualDirectionalDerivativeEvaluation: FunctionExpression, initialGuess: Expression): NonLinearEquationRootExpression
+differentialAlgebraicEquationRoot(equations: Expression, dependentVariables: Expression, independentVariable: Expression, independentVariableValues: Expression): DifferentialAlgebraicEquationRootExpression
+minimum(criterion: Expression, variables: Expression, initialGuess: Expression): MinimumExpression
+value(a: Expression): Expression
```

Figure 70 – Quelques services proposés par le système de calcul symbolico-numérique.

Afin de résumer les services proposés par le système de calcul symbolico-numérique, le diagramme de classes de la Figure 70 rappelle les principales méthodes publiques de la classe **SymbolicNumericalCalculationSystem**. Comme auparavant, afin de ne pas surcharger inutilement l'interface, seuls quelques opérateurs arithmétiques et quelques fonctions mathématiques y figurent.

Annexe B. Réutilisation performante d'expressions mathématiques

Le système de résolution d'équations continues introduit dans la partie 2.2.1 vise un objectif de fiabilité dans les résultats numériques. Les expressions mathématiques requises par les méthodes de résolution du calcul numérique sont construites et simplifiées formellement, avant d'être évaluées numériquement en divers points à la demande du code résolution. L'approximation numérique intervient presque uniquement lors de ces évaluations en virgule flottante. Son influence est donc sans doute moindre que lors d'un calcul complet en virgule flottante. L'utilisation d'expressions analytiques exactes des fonctions dérivées, au lieu des approximations usuelles que sont les schémas aux différences finies, doit contribuer à atteindre l'objectif de fiabilité recherché.

En contrepartie, l'utilisation d'expressions mathématiques formelles a un coût. Celles-ci, plus structurées que les seuls nombres réels ou entiers manipulés dans les environnements de simulation numérique, requièrent a priori plus de place mémoire. En effet, il s'agit de stocker non seulement les données numériques elles-mêmes, mais aussi les informations de structure décrivant l'organisation d'une expression comme une liste ordonnée de sous-expressions à laquelle est appliquée un opérateur arithmétique ou plus généralement une fonction. Manipuler ces expressions mathématiques formelles requiert a priori plus de temps d'exécution puisqu'elles doivent être non seulement évaluées numériquement, mais aussi créées, transformées formellement, et détruites.

Le surcoût en taille mémoire et en temps d'exécution associé à la manipulation d'expressions formelles explique notamment l'approche détaillée en 1.2, consistant à utiliser le calcul formel pour synthétiser des codes de simulation numérique. Le système de simulation et d'optimisation symbolico-numérique présenté se détourne de cette démarche. Dans un même environnement de simulation, la combinaison d'étapes de transformations formelles d'expressions mathématiques, et d'étapes d'exécution d'instructions de calcul numérique est vue comme le meilleur moyen d'atteindre à la fois l'objectif de fiabilité et l'objectif de performance. Les étapes d'exécution d'instructions de calcul numérique visent évidemment l'obtention rapide de résultats exacts. Les étapes de transformations formelles d'expressions mathématiques se justifient d'abord par la recherche de l'exactitude. Un gain en temps de calcul est une conséquence éventuelle d'un gain de précision lié au calcul formel.

Des gains de performance systématiques lors de la manipulation d'expressions mathématiques formelles imposent des mises en œuvre informatiques particulières du système

proposé. Ainsi, afin de limiter, voire de contrebalancer le surcoût en taille mémoire, la partie B.1 formalise-t-elle un processus de création des expressions mathématiques formelles contrôlant la coexistence d'expressions identiques dans le système de calcul. Ce processus s'inspire du mécanisme de partage de sous-expressions adopté par Maple, ce mécanisme étant sans doute le plus abouti parmi ceux proposés par les divers systèmes de calcul formel. La partie B.2 montre tout l'intérêt d'un mécanisme de cache d'évaluation, tirant parti de ce processus original de création des expressions mathématiques formelles afin de limiter, voire de contrebalancer le surcoût en temps d'exécution.

B.1. Contrôle dynamique du partage de sous-expressions

Le système de résolution symbolico-numérique d'équations continues pourrait faire appel à un des systèmes de calcul formel existants pour construire, transformer formellement, et évaluer numériquement les expressions mathématiques formelles recensées dans la partie 2.2.1. Il s'agirait alors de définir un protocole de communication entre un, ou des systèmes de calcul formel et une, ou des bibliothèques mathématiques numériques, puis de traiter l'ordonnancement entre les tâches de calcul numérique et les tâches de calcul formel. Le travail présenté définit un système avec ses propres types de données pour créer des expressions mathématiques formelles, et ses propres traitements pour transformer les expressions créées. Les étapes de calcul numérique sont totalement prises en charge par des bibliothèques mathématiques numériques existantes. A l'inverse, les étapes de calcul formel sont totalement prises en charge par le système en cours de définition. Nous avons ainsi la maîtrise des structures de données du système de calcul formel, en particulier nous contrôlons le mécanisme de création et de destruction des expressions mathématiques formelles. Seul le processus de création des expressions est envisagé ici. La destruction des expressions lorsque celles-ci ne sont plus référencées s'appuie sur un mécanisme original de comptage des références. Le ramasse-miettes associé au système de calcul symbolico-numérique spécifié ici sera, soit intégré nativement à la plate-forme du modèle d'implémentation (dans le cas d'une implémentation au sein de la plate-forme Java par exemple), soit programmé par les développeurs (c'est le cas du prototype eXMSL réalisé, dans lequel les expressions mathématiques sont implantées en FORTRAN 90).

La partie 1.2.4.3 rappelle une technique d'optimisation de code, mise en œuvre par les compilateurs : l'élimination des sous-expressions communes. Le compilateur transforme une séquence d'instructions en une nouvelle séquence d'instructions, où l'évaluation de certaines sous-expressions communes est faite au préalable, et stockée dans des variables intermédiaires. Les instructions initiales sont également transformées : chaque sous-expression commune y est remplacée par la variable temporaire correspondante.

Par analogie avec l'élimination des sous-expressions communes dans les instructions compilées, le partage des sous-expressions communes dans les systèmes de calcul formel évite, ou limite, la coexistence d'expressions identiques parmi les expressions mathématiques formelles représentées en mémoire.

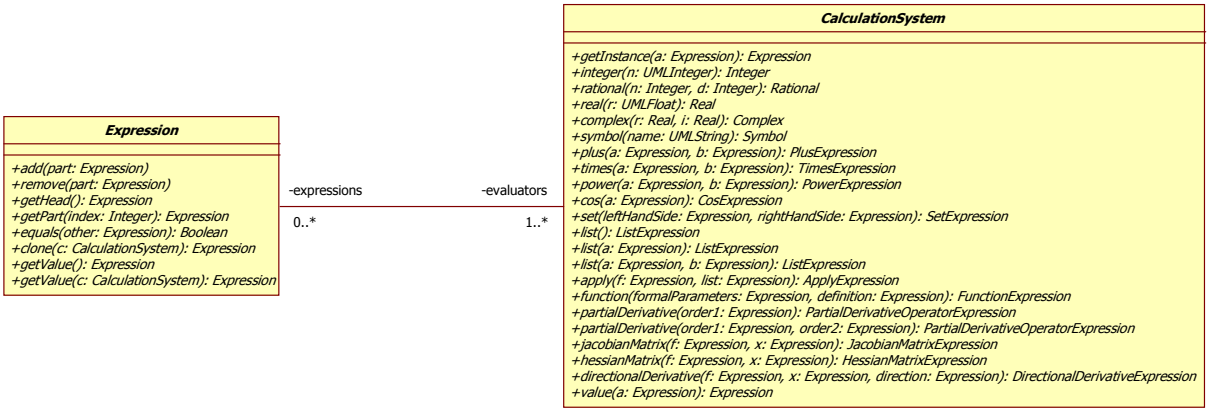

Figure 71 – Modèle structurel d'un système de calcul pour l'évitement des sous-expressions communes.

Afin de formaliser ce processus on suppose tout d'abord que le système de calcul formel maintient une liste exhaustive des expressions qu'il a créées jusqu'alors. Cette liste apparaît dans le diagramme de classes de la Figure 71 comme un attribut privé **expressions** de la classe **CalculationSystem**. Immédiatement après la création d'une expression mathématique formelle à partir des sous-expressions qui la constituent et de la fonction qui leur est appliquée (**plus**, **times**, **set**, …), le système vérifie l'existence simultanée d'une autre expression mathématique identique à l'aide de la méthode **getInstance**. Si tel est le cas, l'expression antérieure est retournée comme résultat de la création, l'expression qui vient d'être créée est détruite. Un tel processus de création évite à tout instant « observable » la coexistence de multiples instances d'une expression mathématique formelle. Contrairement à des systèmes définissant des règles d'élimination de sous-expressions communes tels que DICE dans Ctadel, cité en 1.2.4.3, l'évitement des instances multiples d'une même expression est simple et systématique. Aucune variable temporaire n'est introduite. Il n'y a donc pas à calculer un meilleur compromis entre le coût des opérations d'accès à la mémoire et le coût des opérations arithmétiques. Par construction, l'ensemble des expressions mathématiques formelles manipulées par le système de résolution symbolico-numérique à un instant donné est minimal du point de vue du stockage en mémoire : aucun doublon d'une expression mathématique n'est présent. En particulier, au lieu de référencer plusieurs instances d'une sous-expression commune, les expressions mathématiques formelles référencent une instance unique de la sous-expression commune. Ce processus de création particulier peut donc être vu comme une élimination dynamique et totale des sous-expressions communes.

Par rapport à un processus de création autorisant les doublons, le gain mémoire obtenu en appliquant la stratégie d'évitement des sous-expressions communes semble garanti par construction. En fait, ce gain n'est pas certain. La mise en œuvre de la stratégie d'évitement des sous-expressions communes impose l'existence d'une structure de données supplémentaire inventoriant les expressions déjà créées par le système. L'univers des expressions mathématiques possibles étant infini, une structure de données candidate est une table de hachage dont les valeurs sont les expressions mathématiques formelles, et les clés sont des représentations textuelles uniques et non ambiguës de chaque expression (par exemple la transcription en MathML de contenu de l'expression mathématique). La table de hachage représente donc un surcoût non négligeable de stockage en mémoire vive lors de la simulation. La gestion de cette table de hachage –calcul des codes de hachage, parcours des listes chaînées, insertion ou suppression de valeurs, et redimensionnement de la table- conduit également à un surcoût de temps d'exécution. Le processus de création d'expressions mathématiques formelles présenté semble donc de nouveau conduire à la résolution d'un compromis entre temps de calcul et stockage en mémoire. Le système de calcul doit-il conserver une référence vers toute expression mathématique créée lors d'une session, y compris les résultats de calculs intermédiaires ? Cette stratégie d'évitement des instances multiples d'une même expression doit-elle s'appliquer à tous les types d'expressions mathématiques formelles, ou seulement à certains d'entre eux ?

A ce stade de la discussion une réponse très partielle peut être apportée à la dernière question. Tous les symboles, qu'ils soient prédéfinis par le système, ou définis par l'utilisateur, doivent être uniques en cours de simulation. En effet, outre son nom, à chaque symbole est associé un ensemble de données sémantiques qui précisent, qualifient le symbole. Dans le cas du système de résolution symbolico-numérique discuté ici, ces données sémantiques sont dynamiques et comprennent, sur le modèle du système de calcul formel Mathematica, les définitions dans lesquelles intervient ce symbole, et la liste des propriétés mathématiques (commutativité ou associativité par exemple) lorsque le symbole désigne un opérateur ou une fonction mathématique. Créer de multiples instances d'un symbole et assurer l'identité des données sémantiques dynamiques entre les différentes instances tout au long de la simulation relèverait de l'exercice de style inutile !

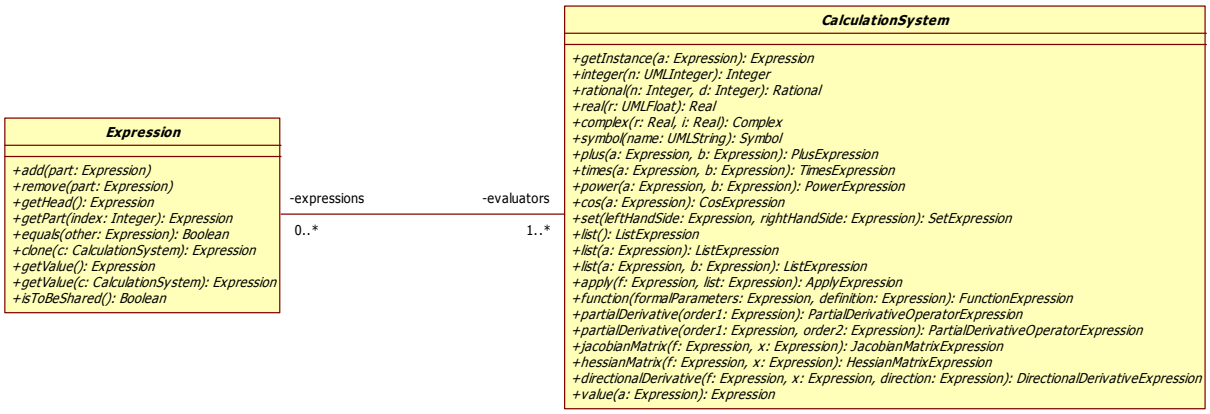

Figure 72 – Modèle structurel d'un système de calcul pour l'évitement des sous-expressions communes selon leurs types.

La Figure 72 formalise les éléments nous permettant d'appliquer une stratégie d'évitement des sous-expressions communes selon leurs types. La méthode **isToBeShared**, déclarée dans la classe **Expression** est définie dans les sous-classes de **Expression** : les classes pour lesquelles la méthode **isToBeShared** retourne la valeur « vrai » donnent naissance à des expressions qui sont systématiquement partagées par le système de calcul, les classes pour lesquelles la méthode **isToBeShared** retourne la valeur « faux » donnent naissance à des expressions dont la référence n'est pas conservée par le système de calcul formel, et qui ne sont donc pas partagées. Selon ce modèle, la classe **Symbol** définit la méthode **isToBeShared** comme une méthode retournant toujours la valeur « vrai ». Sur le même principe, la représentation unique de chaque nombre dans un système de calcul est obtenue aisément en proposant dans la classe **Number** une version concrète de la méthode **isToBeShared** retournant toujours la valeur « vrai ».

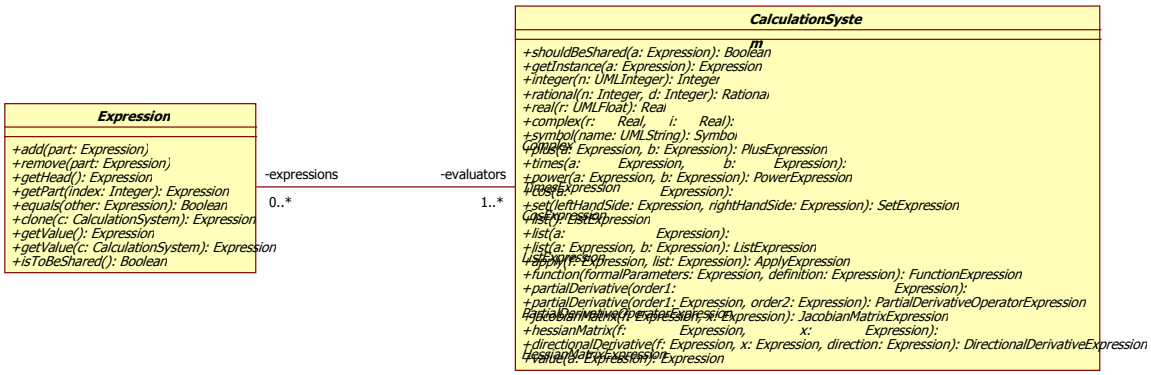

Figure 73 – Modèle structurel d'un système de calcul pour l'évitement des sous-expressions communes selon leurs types, et leurs coûts d'évaluation

La spécification précédente du partage des sous-expressions communes dans un système de calcul semble suffisante si l'on met uniquement en regard du coût du stockage de plusieurs occurrences d'une même expression le temps de calcul requis pour retrouver une expression

partagée par le système. Selon ce point de vue, certaines expressions, dont le type est connu a priori, doivent être partagées. Il s'agit des symboles, des nombres entiers, éventuellement des nombres réels… Toutefois cette spécification ignore un coût primordial, celui de l'évaluation des expressions. Si le système de calcul dispose d'un mécanisme de cache d'évaluation, tel que celui décrit dans la partie 0, le coût du partage d'une expression doit également être mis en regard du coût de réévaluation d'une expression identique par le système. Partager une expression dont l'évaluation est coûteuse a tout son intérêt dès lors que cela évite des évaluations ultérieures d'expressions identiques. Au vu de cette remarque, la Figure 73 propose un modèle de conception plus souple pour le partage des sous-expressions communes. Le système de calcul juge des expressions dont l'évaluation est coûteuse. Si tel est le cas, ce système les référence pour les partager ultérieurement. En pratique, la méthode **shouldBeShared**, déclarée dans la classe **CalculationSystem**, évalue l'intérêt de partager une expression au vu du coût de son évaluation, estimé ou observé, et au vu de la probabilité d'évaluation ultérieure d'une expression identique. Dans le cas où la méthode **shouldBeShared** retourne la valeur « vrai », la méthode appelée ajoute l'expression construite à l'attribut **expressions** de la classe **CalculationSystem**.

Les idées présentées ci-dessus autour du partage de sous-expressions communes dans un système de calcul peuvent être formalisées dans un modèle de conception « partage d'objets identiques », intégrant le modèle de conception « fabrique abstraite » et adaptant le modèle de conception « poids mouche » détaillé dans (Gamma, Helm, Johnson, & Vlissides 1999). Une fabrique abstraite[40] **f** fournit l'accès à des produits au travers de diverses méthodes. Une demande d'accès à un produit donné comprend deux étapes :

1. un objet **o** du type et de la structure souhaités est créé à partir d'un constructeur d'une classe concrète associée à un type de produit particulier ;

2. l'objet créé est transmis comme argument d'appel de la méthode **getInstance** de la classe **ConcreteFactory**. Si un des objets de la collection **products** est identique à **o** alors **getInstance** retourne l'objet **x** partagé. Si **o** ne figure pas dans **products**, ***o*** est éventuellement ajouté à **products** et retourné par la méthode **getInstance**, puis par la méthode d'accès au produit donné.

[40] Le modèle de conception « fabrique abstraite » a été introduit dans la partie 2.1.1.1.

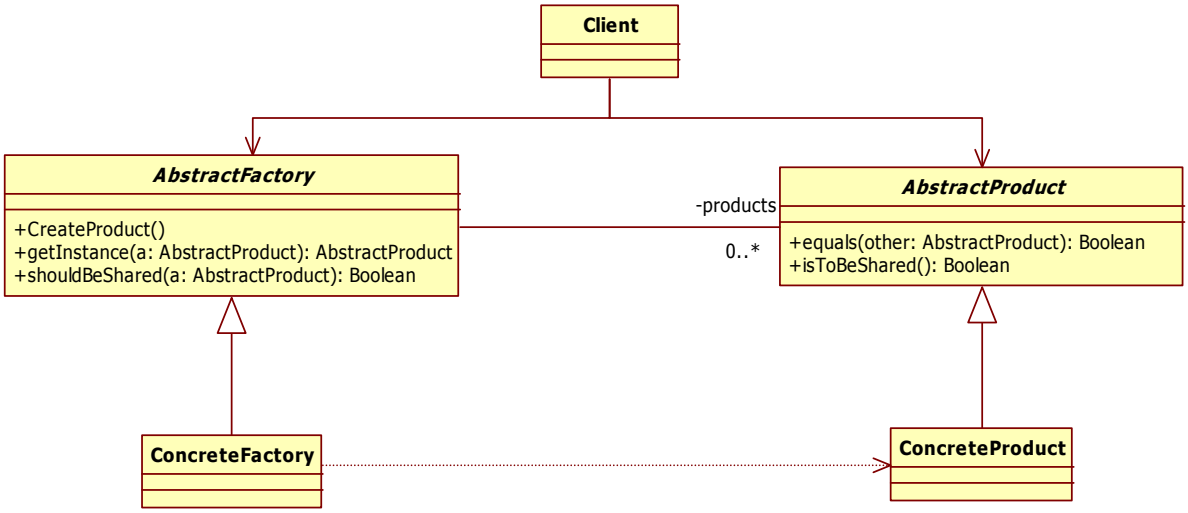

Figure 74 – Modèle de conception « partage d'objets identiques ».

La Figure 74 donne la structure du modèle de conception introduit. Les deux étapes de l'accès à un produit donné constituent le modèle comportemental. Celui-ci varie selon l'algorithme précis de l'action « **o** est éventuellement ajouté à **products** ». Si « **o** est toujours ajouté à **products** », alors aucun doublon d'un produit ne sera jamais fourni par la fabrique. Si « **o** est ajouté à **products** si et seulement si **o.isToBeShared()** », des doublons de l'objet **o** peuvent être fournis par la fabrique uniquement si le concepteur de la classe de base de **o** le choisit. Si « **o** est ajouté à products si et seulement si **(o.isToBeShared() ou f.shouldBeShared(o))** », des doublons de l'objet **o** peuvent être fournis par la fabrique uniquement si le concepteur de la classe de base de **o** le choisit, ou si le concepteur de la classe de base de **f** le choisit.

Dans l'étude du partage des sous-expressions communes dans un système de calcul, les expressions sont les produits du modèle de conception « partage d'objets identiques », tandis que les systèmes de calcul sont les fabriques.

B.2. Réutilisation des résultats d'évaluations précédentes

Les expressions mathématiques formelles sont construites pour être transformées. Même si le résultat n'est pas toujours numérique, l'ensemble des étapes de transformation formelle conduisant à une nouvelle expression mathématique formelle est appelé évaluation. Le processus d'évaluation choisi ici est un processus itératif qui, à chaque itération, tente d'appliquer un certain nombre de transformations formelles à l'expression en entrée pour produire une expression en sortie. Le processus itératif s'interrompt lorsque les expressions en entrée et en sortie sont identiques.

Il semble intéressant de mettre en place un mécanisme dans lequel une expression, une fois évaluée, référence le résultat de son évaluation. Un tel mécanisme de cache d'évaluation a pour but de réutiliser le résultat de calculs précédents, donc de contribuer à l'objectif de

performance. (Jourda, Joulia, & Koehret 1996) décrit l'application d'un cache d'évaluation au sein d'un système d'objets pour la modélisation et la simulation des procédés chimiques. Le résultat d'un calcul de propriétés thermodynamiques par la méthode d'un objet est stocké comme un attribut. Si l'appel suivant de la même méthode, à partir du même objet, comporte des arguments qui n'ont pas été modifiés, la méthode retourne l'attribut sans procéder au moindre calcul.

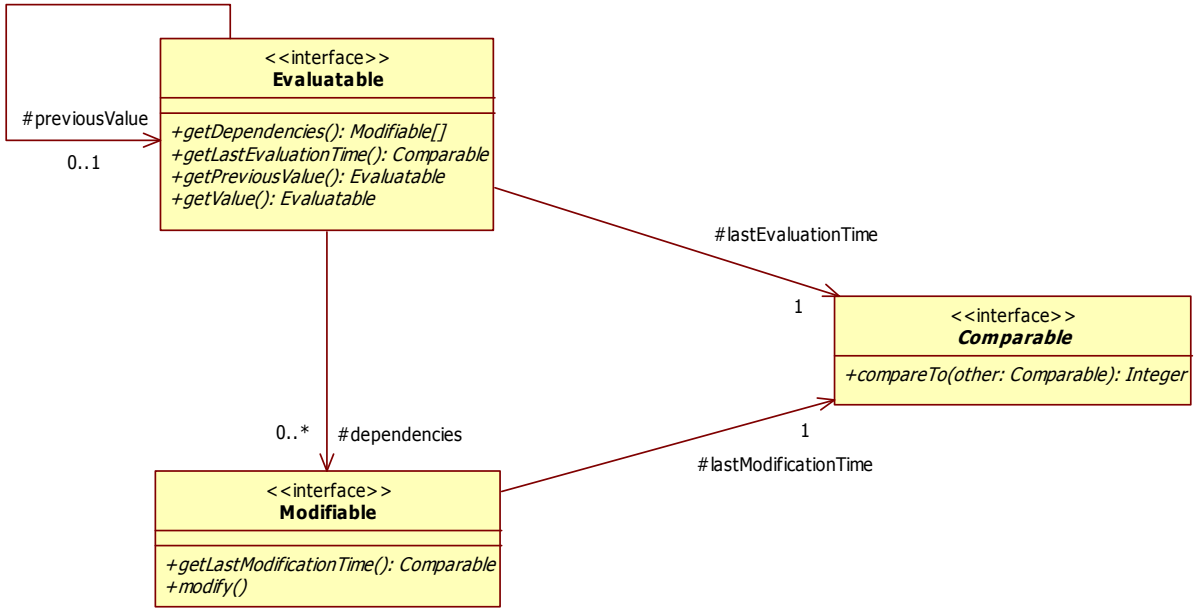

Figure 75 – Modèle de conception « cache d'évaluation ».

Sur les bases du travail cité, une généralisation conduit à proposer le modèle de conception « cache d'évaluation » de la Figure 75, applicable à tout système d'objets dont certains peuvent être modifiés, et dont certains peuvent être évalués. Les classes des objets pouvant être modifiés implémentent l'interface **Modifiable**. Les classes des objets pouvant être évalués implémentent l'interface **Evaluatable**. L'évaluation d'un objet **Evaluatable** produit un autre objet **Evaluatable**, représentant le résultat intermédiaire ou définitif d'un calcul. La méthode **getpreviousValue** retourne le résultat de l'appel précédent à la méthode **getValue**, la plupart du temps stocké dans un attribut jouant le rôle de cache d'évaluation. La date de dernière évaluation d'un objet est comparée aux dates respectives de modification des divers objets dont dépend le résultat d'évaluation. Si les dates de modification de toutes les dépendances sont antérieures à la date de dernière évaluation de l'objet, la valeur du cache d'évaluation est retournée. Dans le cas contraire, la valeur du cache d'évaluation peut être caduque ; un nouveau calcul est nécessaire.

L'implémentation de l'interface **Comparable**[41] permet de définir des événements temporels et de les ordonner.

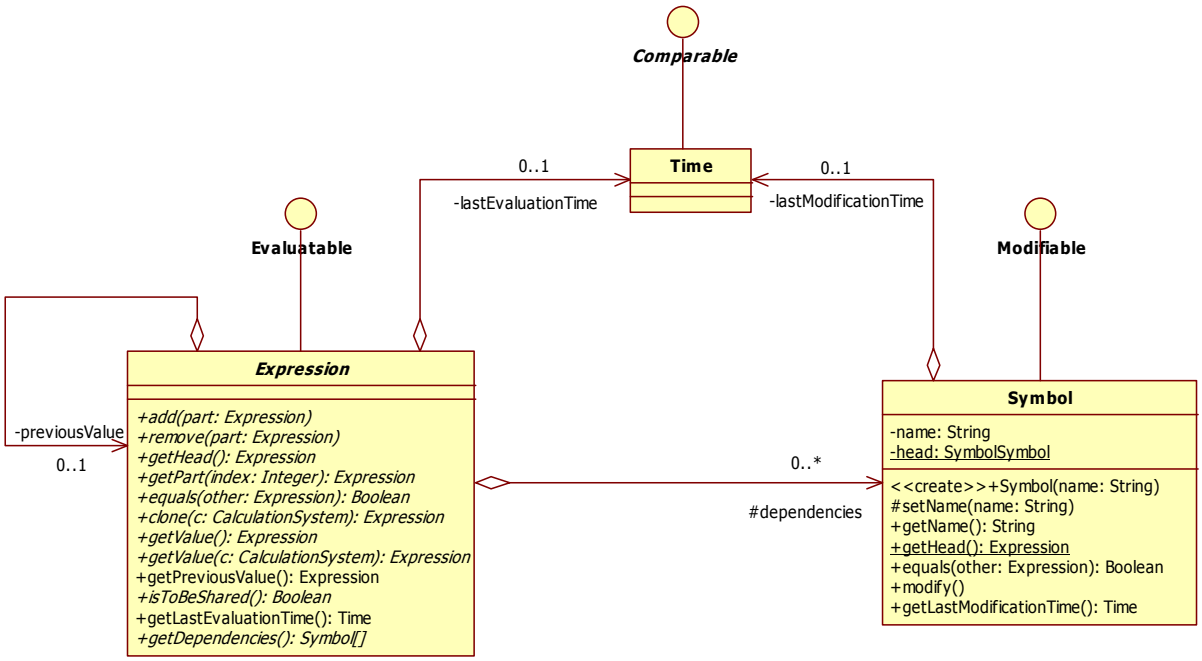

Figure 76 – Modèle de conception « cache d'évaluation » appliqué à la spécification d'un système de calcul.

Dans le cadre de tout système de calcul formel, dédié à l'évaluation d'expressions mathématiques, la classe **Expression** implémente l'interface **Evaluatable** car tout objet **Expression** peut être évalué. Dans le cadre du système de calcul formel que nous spécifions, seuls les symboles peuvent éventuellement être modifiés, lors d'une affectation. En conséquence, seule la classe **Symbol** implémente l'interface **Modifiable**. Cette hypothèse est également adoptée par le système Mathematica, qui permet la modification de valeurs et de propriétés mathématiques associées à des symboles (dénommées attributs). Pourtant, elle ne semble en rien spécifique d'un système de calcul formel particulier. La Figure 76 modifie les classes **Expression** et **Symbol** afin que le système de calcul symbolico-numérique spécifié puisse disposer du mécanisme de cache d'évaluation modélisé ci-dessus. Une classe **Time** implémente la méthode **compareTo** de l'interface **Comparable,** de sorte que les expressions puissent conserver les instants de leurs modifications ou de leurs évaluations.

Couplé à la stratégie d'évitement des instances multiples d'une même expression, le mécanisme de cache d'évaluation proposé apparaît comme un outil efficace pour atteindre l'objectif de performance : le résultat de l'évaluation d'une expression E peut être réutilisé lors de l'évaluation de toute expression E', contenant la sous-expression E, tant qu'une expression à

[41] L'interface **Comparable** s'inspire de l'interface **Comparable** issue de la plate-forme Java.

effet de bord n'a pas été évaluée. En effet, à partir de l'unique expression E, il est possible d'accéder directement au champ référençant le résultat de l'évaluation de E, lorsque celui-ci est disponible et est encore valide.

Suite à l'introduction du mécanisme de cache d'évaluation, il semble judicieux d'appliquer la stratégie d'évitement des instances multiples d'une même expression aux expressions dont l'évaluation est coûteuse, et dont on estime qu'elles seront évaluées plusieurs fois. La non évaluation ultérieure compensera le surcoût en temps de calcul lié à la recherche dans la structure de données choisie d'une autre instance de l'expression. A ce titre, la stratégie d'évitement des instances multiples d'une même expression et le mécanisme de cache d'évaluation sont complémentaires.

Annexe C. Cas d'utilisation de la bibliothèque mathématique eXMSL FORTRAN 90 Library et des composants eXMSL on the Web

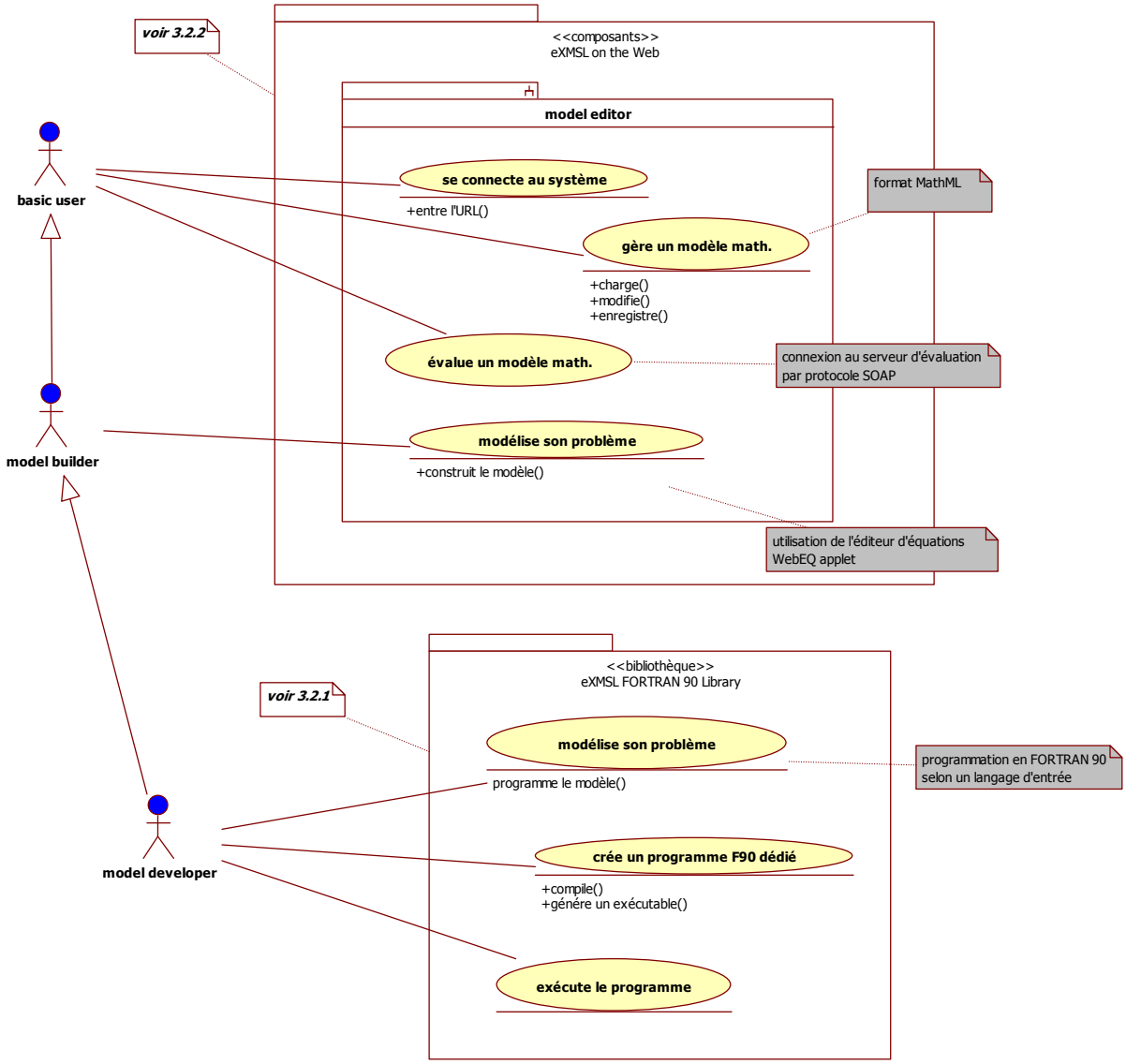

Figure 77 – Cas d'utilisation de la bibliothèque mathématique eXMSL FORTRAN 90 Library et des composants eXMSL on the Web.

Annexe D. Un document MathML produit par eXMSL on the Web

Le document qui suit, au format MathML 2.0, stocke deux expressions mathématiques : la spécification d'une famille de problèmes d'optimisation $\min_{(x,y)\in\Re^2} (1-x)^2 + k(y^2 - x^2)$, et les solutions numériques, calculées par le système eXMSL, aux problèmes respectivement associés aux valeurs $k = 0{,}001$, $k = 0{,}0001$ et $k = 0{,}00001$.

Le document MathML est structuré par les évaluations. A chaque évaluation correspond une balise <semantics>. Chacune de ces balises se subdivise en deux éléments : l'expression mathématique à évaluer, puis le résultat de l'évaluation vu comme une annotation. Les symboles définis par eXMSL, et n'ayant pas de correspondance directe en MathML, sont traduits par des éléments terminaux <csymbol>. Les noms des symboles définis par eXMSL ne sont pas directement utilisés au sein des balises <csymbol>. Dans le but d'échanger directement des portions de documents MathML entre eXMSL et le logiciel Mathematica, les noms des symboles eXMSL sont traduits en leurs correspondants Mathematica lorsque c'est possible. Ainsi, LinearEquationRootSymbol et NonLinearEquationRootSymbol sont-ils tous les deux traduits en FindRoot, algebraicDifferentialEquationRootSymbol en NDSolve, MinimumSymbol en FindMinimum ou NMinimize, et setSymbol en Set ou SetDelayed.

```
<math>

  <semantics>
    <apply xmlns="http://www.w3.org/1998/Math/MathML">
      <ci>Set</ci>
      <ci>k</ci>
      <cn type="real">0.001</cn>
    </apply>
    <annotation-xml encoding="MathML-Content" id="evaluationResult">
      <apply xmlns="http://www.w3.org/1998/Math/MathML">
        <csymbol>SetDelayed</csymbol>
        <apply>
          <csymbol>In</csymbol>
          <cn type="integer">1</cn>
        </apply>
        <apply>
          <csymbol>Set</csymbol>
          <ci>k</ci>
          <cn type="real">0.001</cn>
        </apply>
      </apply>
      <apply xmlns="http://www.w3.org/1998/Math/MathML">
        <csymbol>Set</csymbol>
        <apply>
          <csymbol>Out</csymbol>
          <cn type="integer">1</cn>
        </apply>
        <cn type="real">0.001</cn>
      </apply>
```

```
    </annotation-xml>
  </semantics>

  <semantics>
    <apply xmlns="http://www.w3.org/1998/Math/MathML">
      <ci>FindMinimum</ci>
      <apply>
        <plus/>
        <apply>
          <power/>
          <apply>
            <minus/>
            <cn type="integer">1</cn>
            <ci>x</ci>
          </apply>
          <cn type="integer">2</cn>
        </apply>
        <apply>
          <times/>
          <ci>k</ci>
          <apply>
            <minus/>
            <apply>
              <power/>
              <ci>y</ci>
              <cn type="integer">2</cn>
            </apply>
            <apply>
              <power/>
              <ci>x</ci>
              <cn type="integer">2</cn>
            </apply>
          </apply>
        </apply>
      </apply>
      <list>
        <list>
          <ci>x</ci>
          <cn type="integer">1</cn>
        </list>
        <list>
          <ci>y</ci>
          <cn type="integer">5</cn>
        </list>
      </list>
    </apply>
    <annotation-xml encoding="MathML-Content" id="evaluationResult">
      <apply xmlns="http://www.w3.org/1998/Math/MathML">
        <csymbol>SetDelayed</csymbol>
        <apply>
          <csymbol>In</csymbol>
          <cn type="integer">2</cn>
        </apply>
        <apply>
          <csymbol>FindMinimum</csymbol>
          <apply>
            <plus/>
            <apply>
              <power/>
              <apply>
                <minus/>
                <cn type="integer">1</cn>
                <ci>x</ci>
              </apply>
              <cn type="integer">2</cn>
```

```
            </apply>
            <apply>
              <times/>
              <ci>k</ci>
              <apply>
                <minus/>
                <apply>
                  <power/>
                  <ci>y</ci>
                  <cn type="integer">2</cn>
                </apply>
                <apply>
                  <power/>
                  <ci>x</ci>
                  <cn type="integer">2</cn>
                </apply>
              </apply>
            </apply>
          </apply>
          <list>
            <list>
              <ci>x</ci>
              <cn type="integer">1</cn>
            </list>
            <list>
              <ci>y</ci>
              <cn type="integer">5</cn>
            </list>
          </list>
        </apply>
      </apply>
      <apply xmlns="http://www.w3.org/1998/Math/MathML">
        <csymbol>Set</csymbol>
        <apply>
          <csymbol>Out</csymbol>
          <cn type="integer">2</cn>
        </apply>
        <list>
          <cn type="real">-0.00100100100091755</cn>
          <list>
            <apply>
              <csymbol>Rule</csymbol>
              <ci>x</ci>
              <cn type="real">1.00100105564492</cn>
            </apply>
            <apply>
              <csymbol>Rule</csymbol>
              <ci>y</ci>
              <cn type="real">0.00000897026760047123</cn>
            </apply>
          </list>
        </list>
      </apply>
    </annotation-xml>
  </semantics>

  <semantics>
    <apply xmlns="http://www.w3.org/1998/Math/MathML">
      <ci>Set</ci>
      <ci>k</ci>
      <cn type="real">0.0001</cn>
    </apply>
    <annotation-xml encoding="MathML-Content" id="evaluationResult">
      <apply xmlns="http://www.w3.org/1998/Math/MathML">
        <csymbol>SetDelayed</csymbol>
```

```
        <apply>
          <csymbol>In</csymbol>
          <cn type="integer">3</cn>
        </apply>
        <apply>
          <csymbol>Set</csymbol>
          <ci>k</ci>
          <cn type="real">0.0001</cn>
        </apply>
      </apply>
      <apply xmlns="http://www.w3.org/1998/Math/MathML">
        <csymbol>Set</csymbol>
        <apply>
          <csymbol>Out</csymbol>
          <cn type="integer">3</cn>
        </apply>
        <cn type="real">0.0001</cn>
      </apply>
    </annotation-xml>
  </semantics>

  <semantics>
    <apply xmlns="http://www.w3.org/1998/Math/MathML">
      <ci>FindMinimum</ci>
      <apply>
        <plus/>
        <apply>
          <power/>
          <apply>
            <minus/>
            <cn type="integer">1</cn>
            <ci>x</ci>
          </apply>
          <cn type="integer">2</cn>
        </apply>
        <apply>
          <times/>
          <ci>k</ci>
          <apply>
            <minus/>
            <apply>
              <power/>
              <ci>y</ci>
              <cn type="integer">2</cn>
            </apply>
            <apply>
              <power/>
              <ci>x</ci>
              <cn type="integer">2</cn>
            </apply>
          </apply>
        </apply>
      </apply>
      <list>
        <list>
          <ci>x</ci>
          <cn type="integer">1</cn>
        </list>
        <list>
          <ci>y</ci>
          <cn type="integer">5</cn>
        </list>
      </list>
    </apply>
    <annotation-xml encoding="MathML-Content" id="evaluationResult">
```

```
<apply xmlns="http://www.w3.org/1998/Math/MathML">
  <csymbol>SetDelayed</csymbol>
  <apply>
    <csymbol>In</csymbol>
    <cn type="integer">4</cn>
  </apply>
  <apply>
    <csymbol>FindMinimum</csymbol>
    <apply>
      <plus/>
      <apply>
        <power/>
        <apply>
          <minus/>
          <cn type="integer">1</cn>
          <ci>x</ci>
        </apply>
        <cn type="integer">2</cn>
      </apply>
      <apply>
        <times/>
        <ci>k</ci>
        <apply>
          <minus/>
          <apply>
            <power/>
            <ci>y</ci>
            <cn type="integer">2</cn>
          </apply>
          <apply>
            <power/>
            <ci>x</ci>
            <cn type="integer">2</cn>
          </apply>
        </apply>
      </apply>
    </apply>
    <list>
      <list>
        <ci>x</ci>
        <cn type="integer">1</cn>
      </list>
      <list>
        <ci>y</ci>
        <cn type="integer">5</cn>
      </list>
    </list>
  </apply>
</apply>
<apply xmlns="http://www.w3.org/1998/Math/MathML">
  <csymbol>Set</csymbol>
  <apply>
    <csymbol>Out</csymbol>
    <cn type="integer">4</cn>
  </apply>
  <list>
    <cn type="real">-0.000100010000894667</cn>
    <list>
      <apply>
        <csymbol>Rule</csymbol>
        <ci>x</ci>
        <cn type="real">1.00010001376377</cn>
      </apply>
      <apply>
        <csymbol>Rule</csymbol>
```

```
              <ci>y</ci>
              <cn type="real">-0.0000324682222379294</cn>
            </apply>
          </list>
        </list>
      </apply>
    </annotation-xml>
  </semantics>

  <semantics>
    <apply xmlns="http://www.w3.org/1998/Math/MathML">
      <ci>Set</ci>
      <ci>k</ci>
      <cn type="real">0.00001</cn>
    </apply>
    <annotation-xml encoding="MathML-Content" id="evaluationResult">
      <apply xmlns="http://www.w3.org/1998/Math/MathML">
        <csymbol>SetDelayed</csymbol>
        <apply>
          <csymbol>In</csymbol>
          <cn type="integer">5</cn>
        </apply>
        <apply>
          <csymbol>Set</csymbol>
          <ci>k</ci>
          <cn type="real">0.00001</cn>
        </apply>
      </apply>
      <apply xmlns="http://www.w3.org/1998/Math/MathML">
        <csymbol>Set</csymbol>
        <apply>
          <csymbol>Out</csymbol>
          <cn type="integer">5</cn>
        </apply>
        <cn type="real">0.00001</cn>
      </apply>
    </annotation-xml>
  </semantics>

  <semantics>
    <apply xmlns="http://www.w3.org/1998/Math/MathML">
      <ci>FindMinimum</ci>
      <apply>
        <plus/>
        <apply>
          <power/>
          <apply>
            <minus/>
            <cn type="integer">1</cn>
            <ci>x</ci>
          </apply>
          <cn type="integer">2</cn>
        </apply>
        <apply>
          <times/>
          <ci>k</ci>
          <apply>
            <minus/>
            <apply>
              <power/>
              <ci>y</ci>
              <cn type="integer">2</cn>
            </apply>
            <apply>
              <power/>
```

```
              <ci>x</ci>
              <cn type="integer">2</cn>
            </apply>
          </apply>
        </apply>
      </apply>
      <list>
        <list>
          <ci>x</ci>
          <cn type="integer">1</cn>
        </list>
        <list>
          <ci>y</ci>
          <cn type="integer">5</cn>
        </list>
      </list>
    </apply>
    <annotation-xml encoding="MathML-Content" id="evaluationResult">
      <apply xmlns="http://www.w3.org/1998/Math/MathML">
        <csymbol>SetDelayed</csymbol>
        <apply>
          <csymbol>In</csymbol>
          <cn type="integer">6</cn>
        </apply>
        <apply>
          <csymbol>FindMinimum</csymbol>
          <apply>
            <plus/>
            <apply>
              <power/>
              <apply>
                <minus/>
                <cn type="integer">1</cn>
                <ci>x</ci>
              </apply>
              <cn type="integer">2</cn>
            </apply>
            <apply>
              <times/>
              <ci>k</ci>
              <apply>
                <minus/>
                <apply>
                  <power/>
                  <ci>y</ci>
                  <cn type="integer">2</cn>
                </apply>
                <apply>
                  <power/>
                  <ci>x</ci>
                  <cn type="integer">2</cn>
                </apply>
              </apply>
            </apply>
          </apply>
          <list>
            <list>
              <ci>x</ci>
              <cn type="integer">1</cn>
            </list>
            <list>
              <ci>y</ci>
              <cn type="integer">5</cn>
            </list>
          </list>
```

```
        </apply>
      </apply>
      <apply xmlns="http://www.w3.org/1998/Math/MathML">
        <csymbol>Set</csymbol>
        <apply>
          <csymbol>Out</csymbol>
          <cn type="integer">6</cn>
        </apply>
        <list>
          <cn type="real">-0.0000100000966189432</cn>
          <list>
            <apply>
              <csymbol>Rule</csymbol>
              <ci>x</ci>
              <cn type="real">1.00000999531785</cn>
            </apply>
            <apply>
              <csymbol>Rule</csymbol>
              <ci>y</ci>
              <cn type="real">0.000581552564351705</cn>
            </apply>
          </list>
        </list>
      </apply>
    </annotation-xml>
  </semantics>

  <semantics class="Symbol">Symbols</semantics>
  <semantics class="Symbol"/>
  <semantics class="Symbol"/>
  <semantics class="Expression">Expressions</semantics>
  <semantics class="Expression"/>
  <semantics class="Expression"/>
</math>
```


Figure 78 – Document MathML produit par eXMSL on the Web lors des optimisations $\min_{(x,y)\in\Re^2} (1-x)^2 + k(y^2 - x^2)$, pour $k = 0{,}001$, $k = 0{,}0001$ et $k = 0{,}00001$.

Akers, R. L., Baffes, P., Kant, E., Randall, C., Steinberg, S., & Young, R. L. 1998, "Automatic synthesis of numerical codes for solving partial differential equations", *Mathematics and Computers in Simulation*, vol. 45, no. 1-2, pp. 3-22.

Alfradique, M. F. & Castier, M. 2005, "Modeling and simulation of reactive distillation columns using computer algebra", *Computers & Chemical Engineering*, vol. 29, no. 9, pp. 1875-1884.

Alloula, K., Belaud, J.-P., & Joulia, X. 2000, "Applying automatic differentiation to computer-aided process engineering software", in *AD 2000 - From Simulation to Optimization - 3rd International Conference on Automatic Differentiation*, Atlas Conferences Inc..

Alloula, K., Belaud, J.-P., & Le Lann, J.-M. 2004, "eXtended Mathematical Software Library for CAPE", *Computer-Aided Chemical Engineering*, vol. 18, pp. 1015-1020.

Alloula, K., Belaud, J.-P., & Le Lann, J.-M. 2007, "Mixing computer algebra and numerical methods when solving CAPE models", *Computer-Aided Chemical Engineering*, vol. 24, pp. 135-140.

Alloula, K., Belaud, J.-P., Leibovici, C., & Le Lann, J.-M. 2006, "Traitement symbolique et numérique de modèles thermodynamiques implicites", in *SIMO 2006*.

Amberg, G., Tonhardt, R., & Winkler, C. 1999, "Finite element simulations using symbolic computing", *Mathematics and Computers in Simulation*, vol. 49, no. 4-5, pp. 257-274.

Association Française d'Ingénierie Système 2004, *Glossaire de base de l'ingénierie de systèmes v1.2*.

Ausbrooks, R., Buswell, S., Carlisle, D., & Dalmas, S. Mathematical Markup Language (MathML) Version 2.0 (Second Edition)
W3C Recommendation 21 October 2003. http://www.w3.org/TR/MathML2/ . 21-10-2003.
Ref Type: Electronic Citation

Barker, H. A. & Zhuang, M. 1997, "Control system analysis using Mathematica and a graphical user interface", *Computing & Control Engineering Journal*, vol. 8, no. 2, pp. 64-69.

Barrère, R. Can Computer Algebra be Liberated from its Algebraic Yoke ? An Algorithmic Approach to Functional Analysis from a Functional Approach to Computing. http://arxiv.org . 3-2-2005.
Ref Type: Electronic Citation

Barrett, W. M., Pons, M., von Wedel, L., & Braunschweig, B. 2007, "An Overview of the Interoperability Roadmap for COM/.NET-Based CAPE-OPEN", *Computer-Aided Chemical Engineering*, vol. 24, pp. 165-170.

Bastos, J. & Monti, A. 2005, "Automatically building customized circuit-based simulation models using symbolic computing", *Mathematics and Computers in Simulation*, vol. 70, no. 4, pp. 203-220.

Baur, W. & Strassen, V. 1983, "The complexity of partial derivatives", *Theoretical Computer Science*, vol. 22, no. 3, pp. 317-330.

Belaud, J.-P., Alloula, K., Le Lann, J.-M., & Joulia, X. 2001, "Open software architecture for numerical solvers : design, implementation and validation", *Computer-Aided Chemical Engineering*, vol. 9, pp. 967-972.

Belaud, J.-P. & Braunschweig, B. 2002, "Making CAPE tools – The CAPE-OPEN Standard : Motivations, development process, technical architecture and examples," in *Software architectures and tools for computer aided process engineering*, R. Gani & B. Braunschweig, eds., Elsevier, pp. 303-322.

Belaud, J.-P., Braunschweig, B., & Pons, M. 2002, "Open software architecture for process simulation", *Computer-Aided Chemical Engineering*, vol. 10, pp. 847-852.

Berzins, M., Dew, P. M., & Furzeland, R. M. 1989, "Developing software for time-dependent problems using the method of lines and differential-algebraic integrators", *Applied Numerical Mathematics*, vol. 5, no. 5, pp. 375-397.

Bischof, C., Carle, A., Corliss, G., & Griewank, A. "ADIFOR: Automatic differentiation in a source translator environment", ACM Press, Berkeley, California, United States, pp. 294-302.

Borst, W. N., Goldman, V. V., & van Hulzen, J. A. "GENTRAN 90: a REDUCE package for the generation of Fortran 90 code", ACM Press, Oxford, United Kingdom, pp. 45-51.

Braun, S. 2000, "Application of computer-algebra simulation (CALS) in industry", *Mathematics and Computers in Simulation*, vol. 53, no. 4-6, pp. 249-257.

Braunschweig, B. 2002, *Vers la Simulation Numérique par Agents Apprenants - Segmentation, dynamisation et autres lois d'évolution des logiciels,* Habilitation à Diriger les Recherches, Université Pierre et Marie Curie, Paris.

Brenan, K. E., Campbell, S. L., & Petzold, L. R. 1996, *Numerical Solution of Initial-Value Problems in Differential-Algebraic Equations*, 2nd edn, SIAM.

Brezinski, C. 2004, "Les fractions continues ou l' art de l' approximation; Continuous fractions or the state of approximation", *Pour la science* no. 316, pp. 64-71.

Brice, C. W., Gödkere, L. U., & Dougal, R. A. 1998, "The Virtual Test Bed: An Environment for Virtual Prototyping", in *International Conference on Electric Ship*, Istanbul, Turkey, pp. 27-31.

Brown, P. N., Hindmarsh, A. C., & Petzold, L. R. 1998, "Consistent Initial Condition Calculation for Differential-Algebraic Systems", *SIAM Journal on Scientific Computing*, vol. 19, no. 5, pp. 1495-1512.

Bunnin, F. O., Guo, Y., Ren, Y., & Darlington, J. 2000, "Design of high performance financial modelling environment", *Parallel Computing*, vol. 26, no. 5, pp. 601-622.

Bunus, P. & Fritzson, P. 2004, "Automated Static Analysis of Equation-Based Components", *Simulation*, vol. 80, no. 7-8, pp. 321-345.

Campbell, S. L. 1986, "Consistent initial conditions for linear time varying singular systems," in *Frequency domain and state space methods for linear systems*, C. I. Byrnes & Lindquist A., eds., Elsevier Science (North-Holland), Amsterdam.

Campbell, S. L. & Griepentrog, E. 1995, "Solvability of General Differential Algebraic Equations", *SIAM Journal on Scientific Computing*, vol. 16, no. 2, pp. 257-270.

Campbell, S. L., Kelley, C. T., & Yeomans, K. D. "Consistent Initial Conditions for Unstructured Higher Index DAEs: A Computational Study", in *Computational Engineering in Systems Applications*, pp. 416-421.

Castier, M. 1999, "Automatic implementation of thermodynamic models using computer algebra", *Computers & Chemical Engineering*, vol. 23, no. 9, pp. 1229-1245.

Chancelier, J. Ph., Gomez, C., & Quadrat, J.-P. 1987, "MACROFORT, A Fortran program generator", *Macsyma letters*.

Chernukhin, Y., Polenov, M., Chandrasekhar, V., Solodovnik, E., Mantooth, H. A., & Dougal, R. A. 2005, "Deploying Modelica models into multiple simulation environments", in *Behavioral Modeling and Simulation Workshop, 2005. BMAS 2005.*, pp. 134-139.

CO-LaN. The CAPE-OPEN Laboratory Network. http://www.colan.org . 2007.
Ref Type: Electronic Citation

COMSOL AB. 2001, *FEMLAB user's guide and introduction : for use with Matlab, Windows, Unix, Linux, Macintosh : version 2.2*
1 COMSOL, Stockholm, Sweden.

Creutzig, C. & Oevel, W. 2004, *MuPAD Tutorial*, 2nd edn, Springer, New-York.

Dall'Osso, A. 2003, "Using computer algebra systems in the development of scientific computer codes", *Future Generation Computer Systems*, vol. 19, no. 2, pp. 143-160.

Dall'Osso, A. 2006, "Computer algebra systems as mathematical optimizing compilers", *Science of Computer Programming*, vol. 59, no. 3, pp. 250-273.

Davenport, J. H., Siret, Y., & Tournier, E. 1987, *Calcul Formel: systèmes et algorithmes de manipulations algébriques* Masson.

de Pelegrini Soares, R. & Secchi, A. R. 2007, "Debugging for Equation-Oriented CAPE Tools", *Computer-Aided Chemical Engineering*, vol. 24, pp. 237-242.

Decyk, V. K., Norton, C. D., & Szymanski, B. K. 1998, "How to support inheritance and run-time polymorphism in Fortran 90", *Computer Physics Communications*, vol. 115, no. 1, pp. 9-17.

Dedieu, J.-P. 2006, *Points fixes, zéros et la méthode de Newton* Springer Berlin Heidelberg.

Dempsey, M. "Dymola for Multi-Engineering Modelling and Simulation", in *Vehicle Power and Propulsion Conference, 2006. VPPC '06. IEEE*, pp. 1-6.

Elmqvist, H., Mattsson, S. E., & Otter, M. 1999, "Modelica-a language for physical system modeling, visualization and interaction", in *Computer Aided Control System Design, 1999*, pp. 630-639.

Engels, H. 1990, *Phase Equilibria and Phase Diagrams of Electrolytes* Dechema.

Fagan, M., Hascoet, L., & Utke, J. 2006, "Data Representation Alternatives in Semantically Augmented Numerical Models", in *Source Code Analysis and Manipulation, 2006. SCAM '06*, pp. 85-94.

Faure, C. 2005, "An automatic differentiation platform: Odyssee", *Future Generation Computer Systems*, vol. 21, no. 8, pp. 1391-1400.

Faure, C., Davenport, J. H., & Naciri, H. 2000, "Multi-valued computer algebra. Calcul formel multi-valué", *Rapport de recherche INRIA* no. 4001, p. 36.

Fousse, L., Hanrot, G., Lefèvre, V., Pélissier, P., & Zimmermann, P. 2007, "MPFR: A multiple-precision binary floating-point library with correct rounding", *ACM Transactions on Mathematical Software*, vol. 33, no. 2.

Fritzson, P. & Engelson, V. 1998, "Modelica - A unified object-oriented language for system modeling and simulation," pp. 67-90.

Gamma, E., Helm, R., Johnson, R., & Vlissides, J. 1999, *Design patterns Catalogue de modèles de conception réutilisables* Vuibert, Paris.

Gates, B. L. 1985, "Gentran: an automatic code generation facility for REDUCE", *ACM SIGSAM Bulletin*, vol. 19, no. 3, pp. 24-42.

Gear, C. 1971, "Simultaneous Numerical Solution of Differential-Algebraic Equations", *Circuits and Systems, IEEE Transactions on [legacy, pre - 1988]*, vol. 18, no. 1, pp. 89-95.

Giusti, M., Hagele, K., Lecerf, G., Marchand, J., & Salvy, B. 2000, "The Projective Noether Maple Package: Computing the Dimension of a Projective Variety", *Journal of Symbolic Computation*, vol. 30, no. 3, pp. 291-307.

Gomez, C. 1990, "MACROFORT: a FORTRAN Code Generator in MAPLE", *Rapport Technique INRIA* no. 0119.

Gomez, C. Systèmes dynamiques. http://www-rocq.inria.fr . 2005.
Ref Type: Electronic Citation

Gomez, C. & Quadrat, J.-P. 1991, "Calcul formel et calcul numérique", *Bulletin de liaison de la recherche en informatique et en automatique* no. 130, pp. 7-9.

Gomez, C. & Scott, T. 1998, "Maple programs for generating efficient FORTRAN code for serial and vectorised machines", *Computer Physics Communications*, vol. 115, no. 2-3, pp. 548-562.

Gräbe, H.-G. 1999, "Algebraic Numbers in Symbolic Computations.", *International Journal of Computer Algebra in Mathematics Education*, vol. 6, pp. 65-84.

Griewank, A., Juedes, D., & Utke, J. 1996, "Algorithm 755: ADOL-C: a package for the automatic differentiation of algorithms written in C/C++", *ACM Transactions on Mathematical Software*, vol. 22, no. 2, pp. 131-167.

Hall, C. A. & Porsching, T. A. 1990, *Numerical analysis of partial differential equations 6* Prentice Hall, Englewood Cliffs, N.J.

Hirsch, M. W. & Smale, S. 1979, "On Algorithms for Solving f(x)=0", *Communications on Pure and Applied Mathematics*, vol. 32, no. 3, pp. 281-312.

Houstis, E. N. & Rice, J. R. 1995, *Multidisciplinary Problem Solving Environments*, Purdue University.

Houstis, E. N. & Rice, J. R. 2000, "Future problem solving environments for computational science", *Mathematics and Computers in Simulation*, vol. 54, no. 4-5, pp. 243-257.

Husa, S., Hinder, I., & Lechner, C. 2006, "Kranc: a Mathematica package to generate numerical codes for tensorial evolution equations", *Computer Physics Communications*, vol. 174, no. 12, pp. 983-1004.

Jenks, R. D. & Sutor, R. S. 1992, *AXIOM the scientific computation system* Springer, New York.

Joulia, X., Alloula, K., & Belaud, J.-P. 1999, *AD-CAPE Project No 24023 - Automatic Differentiation for Computer Aided Process Engineering - Deliverable 3.2.1.a - Report : SMS AD strategy report*, Information Technologies Programme 1994-1998 (ESPRIT IV) - Domain 1: Software Technologies - Task 1.32: Trial Applications.

Joulia, X., Koehret, B., & Enjalbert, M. 1985, "Simulateur modulaire séquentiel à convergence simultanée", *The Chemical Engineering Journal*, vol. 30, no. 3, pp. 113-127.

Jourda, L., Joulia, X., & Koehret, B. 1996, "Introducing ATOM, the Applied Thermodynamics Object-oriented Model", *Computers & Chemical Engineering*, vol. 20, no. Supplement 1, p. S157-S164.

Korelc, J. "Hybrid system for multi-language and multi-environment generation of numerical codes", ACM Press, London, Ontario, Canada, pp. 209-216.

Kunkel, P. & Mehrmann, V. 1998, "Regular solutions of nonlinear differential-algebraic equations and their numerical determination", *Numerische Mathematik*, vol. 79, no. 4, pp. 581-600.

Le Lann, J.-M. 1999, *Des mathématiques à la Simulation Dynamique robuste des Procédés: le traitement algébro-différentiel des équations E.D.A.,* Habilitation à Diriger les Recherches, Institut National Polytechnique de Toulouse.

Mattsson, S. E., Elmqvist, H., & Otter, M. 1998, "Physical system modeling with Modelica", *Control Engineering Practice*, vol. 6, no. 4, pp. 501-510.

McCarl, B. A. McCarl GAMS User Guide - Version 22.5. http://www.gams.com . 2007.
Ref Type: Electronic Citation

Ménissier-Morain, V. 2005, "Arbitrary precision real arithmetic: design and algorithms", *Journal of Logic and Algebraic Programming*, vol. 64, no. 1, pp. 13-39.

Mili, H., Mili, A., Yacoub, S., & Addy, E. 2002, *Reuse Based Software Engineering: Techniques, Organizations, and Measurement* John Wiley & Sons, New York.

Mischler, C., Joulia, X., Hassold, E., Galligo, A., & Esposito, R. "Automatic Differentiation applications to computer aided process engineering", *Computers & Chemical Engineering*, vol. 19, no. Supplement 1, pp. 779-784.

Modelica Association. Modelica - A Unified Object-Oriented Language for Physical Systems Modeling. Language Specification. http://www.modelica.org . 2-2-2005.
Ref Type: Electronic Citation

Monagan, M. B., Geddes, K. O., Heal, K. M., Labahn, G., Vorkoetter, S. M., McCarron, J., & DeMarco, P. 2005a, *Maple 10 Advanced Programming Guide* Maplesoft, Waterloo, Ontario, Canada.

Monagan, M. B., Geddes, K. O., Heal, K. M., Labahn, G., Vorkoetter, S. M., McCarron, J., & DeMarco, P. 2005b, *Maple 10 Introductory Programming Guide* Maplesoft, Waterloo, Ontario, Canada.

Mourrain, B., Vrahatis, M.-N., & Yakoubsohn, J.-C. 2001, "Isolation of real roots and computation of the topological degree. Isolation de racines réelles et calcul du degré topologique.", *Rapport de recherche INRIA* no. 4300.

Müller, N. Th. 2001, "The iRRAM: Exact Arithmetic in C++", *Lecture Notes in Computer Science*, vol. 2064.

Nickel, K. 1985, "Interval mathematics 1985", in *Interval mathematics 1985*, Springer-Verlag, London,UK.

Object Management Group. Unified Modeling Language (UML), version 2.1.1. http://www.omg.org . 3-2-2007.
Ref Type: Electronic Citation

Oh, M. & Pantelides, C. C. 1996, "A modelling and simulation language for combined lumped and distributed parameter systems", *Computers & Chemical Engineering*, vol. 20, no. 6-7, pp. 611-633.

Pantelides, C. C. 1988, "The Consistent Initialization of Differential-Algebraic Systems", *SIAM Journal on Scientific and Statistical Computing*, vol. 9, no. 2, pp. 213-231.

Pop, A., Savga, I., Aßmann, U., & Fritzson, P. 2005, "Composition of XML Dialects: A ModelicaXML Case Study", *Electronic Notes in Theoretical Computer Science*, vol. 114, pp. 137-152.

Process Systems Enterprise Ltd. 2000, *gPROMS advanced user's guide.* http://www.psenterprise.com.

Randall, C. & Kant, E. "Numerical options models without programming", in *Computational Intelligence for Financial Engineering (CIFEr), 1997*, pp. 15-21.

Rascol, E., Meyer, M., Le Lann, J.-M., & Prevost, M. 1998, "Numerical problems encountered in the simulation of reactive absorption: DAE index reduction and consistent initialisation", *Computers & Chemical Engineering*, vol. 22, no. Supplement 1, p. S929-S932.

Recio, T. & Gonzalez-Lopez, M. J. 1998, "Does Computer Algebra help at all learning about real numbers?", *Mathematics and Computers in Simulation*, vol. 45, no. 1-2, pp. 185-195.

Revol, N. & Rouillier, F. 2007, "Motivations for an arbitrary precision interval arithmetic and the MPFI library", *Reliable Computing*, vol. 11, pp. 1-16.

Rioboo, R. 2003, "Towards faster real algebraic numbers", *Journal of Symbolic Computation*, vol. 36, no. 3-4, pp. 513-533.

Saad, Y. & Schultz, M. H. 1986, "GMRES: A Generalized Minimal Residual Algorithm for Solving Nonsymmetric Linear Systems", *SIAM Journal on Scientific and Statistical Computing*, vol. 7, no. 3, pp. 856-869.

Scott, J. A. 2001, "The design of a portable parallel frontal solver for chemical process engineering problems", *Computers & Chemical Engineering*, vol. 25, no. 11-12, pp. 1699-1709.

Sofroniou, M. & Spaletta, G. 2005, "Precise numerical computation", *Journal of Logic and Algebraic Programming*, vol. 64, no. 1, pp. 113-134.

Straka, C. W. 2005, "ADF95: Tool for automatic differentiation of a FORTRAN code designed for large numbers of independent variables", *Computer Physics Communications*, vol. 168, no. 2, pp. 123-139.

Taylor, R. 1997, "Automatic derivation of thermodynamic property functions using computer algebra", *Fluid Phase Equilibria*, vol. 129, no. 1-2, pp. 37-47.

Tijskens, E., Roose, D., Ramon, H., & De Baerdemaeker, J. 2004, "FastDer++, efficient automatic differentiation for non-linear PDE solvers", *Mathematics and Computers in Simulation*, vol. 65, no. 1-2, pp. 177-190.

Trefethen, A. E. & Ford, B. 2000, "Numerical algorithm delivery mechanisms", *Mathematics and Computers in Simulation*, vol. 54, no. 4-5, pp. 259-268.

Vacher, A. & Guittard, P. "Interoperability concept in a COM thermodynamic server architecture. Example of integration in Microsoft® Excel™", in *SIMO 2002*.

van Engelen, R., Wolters, L., & Cats, G. "CTADEL: a generator of multi-platform high performance codes for PDE-based scientific applications", ACM Press, Philadelphia, Pennsylvania, United States, pp. 86-93.

van Engelen, R., Wolters, L., & Cats, G. "PDE-oriented language compilation and optimization with CTADEL for parallel computing", in *Second International Workshop on High-Level Programming Models and Supportive Environments, 1997.*, pp. 105-109.

von Hippel, G. M. 2006, "TaylUR, an arbitrary-order diagonal automatic differentiation package for Fortran 95", *Computer Physics Communications*, vol. 174, no. 7, pp. 569-576.

Wendlandt, J. M. & Marsden, J. E. 1997, "Mechanical integrators derived from a discrete variational principle", *Physica D: Nonlinear Phenomena*, vol. 106, no. 3-4, pp. 223-246.

Whiting, P. G. & Pascoe, R. S. V. 1994, "A history of data-flow languages", *Annals of the History of Computing, IEEE*, vol. 16, no. 4, pp. 38-59.

Wolfe, P. 1982, "Checking the Calculation of Gradients", *ACM Transactions on Mathematical Software*, vol. 8, no. 4, pp. 337-343.

Wolfram, S. 2003, *The mathematica book*, 5th edn, Wolfram Media, Champaign, IL.

Zaraté, P. 2005, *Des Systèmes Interactifs d'Aide à la Décision Aux Systèmes Coopératifs d'Aide à la Décision : Contributions conceptuelles et fonctionnelles,* Habilitation à Diriger les Recherches, Institut National Polytechnique de Toulouse.

Zualkernan, I. A. & Tsai, W. T. "Are knowledge representations the answer to requirement analysis?", in *International Conference on Computer Languages, 1988.*, pp. 437-443.

Thèse de M Karim Alloula

Modèle de coopération entre calcul formel et calcul numérique pour la simulation et l'optimisation des systèmes.

Après avoir étudié les collaborations établies aujourd'hui entre différents environnements de résolution de problèmes, le manuscrit propose un modèle de conception d'un système de calcul basé sur la coopération entre calcul formel et numérique. Cette coopération entre différents sous-systèmes de calcul est de type complémentaire : les rôles sont définis a priori.

Suivant une démarche orientée modèle, le modèle de coopération est spécifié en UML 2.0 selon la vue structurelle et la vue comportementale. A partir du modèle conceptuel, nous définissons les règles de transformation pour produire le modèle d'implémentation spécifique de la « plate-forme » FORTRAN 90.

Au vu des résultats d'études particulières en génie des procédés -la solvatation d'acides forts, et la distillation de Rayleigh- il apparaît que la démarche de calcul coopératif proposée :

- apporte plus d'expressivité lors de la modélisation;
- incite à modéliser les systèmes physiques à l'aide de fonctions, souvent implicites ;
- permet la réutilisation de modèles par la composition et l'assemblage de fonctions ;
- et apporte plus de fiabilité lors de la simulation, notamment grâce au calcul précis des dérivées des modèles.

Mots-clés : Calcul Formel, Calcul Numérique, Modèle de Coopération, Systèmes Industriels, Génie des Procédés.

A co-operative model combining computer algebra and numerical calculation for system simulation and optimization.

After investigating state-of-the-art collaborations between various environments aimed at system resolution, this paper presents a design model for a calculation system based on the co-operation of formal and numerical calculations. This co-operation between multiple sub-systems is complementary: the roles are defined a priori.

Following an object-oriented approach, the model is specified via UML 2.0, in terms of the structural and behavioural views. From the conceptual model, we define the transformation rules required to create the implementation-specific model for the FORTRAN 90 platform.

From the results witnessed within specific process engineering studies - namely the solvation of strong acids and Rayleigh's distillation - it can be seen that this co-operative approach:

- empowers us with an improved expressivity at the modeling stage;
- instigates physical modeling using (often implicit) functions;
- allows model re-use through function aggregation and assembly;
- and brings a greater reliability during simulation, notably as a result of the precise calculation of derivatives of the model.

Keywords: Computer Algebra, Numerical Calculation, Co-operation Model, Industrial Systems, Process Engineering.