Un Logiciel pour le Calcul Formel Distribué

Stéphane Dalmas Marc Gaëtano Alain Sausse
INRIA Sophia Antipolis, projet SAFIR
2004 route des lucioles BP 93
06902 Sophia-Antipolis CEDEX
France
E-mail: {stephane.dalmas, marc.gaetano, alain.sausse}@sophia.inria.fr

Mots clés : environnement pour le calcul formel, calcul symbolique distribué, géométrie algébrique élémentaire.

Résumé

Dans ce papier, nous décrivons le *Central Control*, un composant logiciel qui permet par un échange de données, une collaboration et une coopération entre plusieurs systèmes de calcul symbolique. Dans une seconde partie, nous présentons une bibliothèque qui fournit à un utilisateur du *Central Control* les opérations élémentaires sur les idéaux polynomiaux, mais aussi, qui donne un exemple de programmation sous le *Central Control*.

Abstract

In this paper, we describe the *Central Control*, a software component that enables several symbolic systems to cooperate and exchange data. The Central Control has been designed to be the kernel of an environment for scientific computations which can offer a common and concurrent access to several tools needed by the scientist and the engineer: general purpose and specialized computer algebra systems, visualization tools, links with numerical librairies and tools to manipulate numerical programs. The user can interact with the Central Control through one or more (graphical) user interfaces. The Central Control is a *Scheme* interpreter extended with new primitive types and procedures. In a second part, we present a package which provides to an user of the *Central Control*, the elementary operations on polynomial ideals, but also which gives an example of programming under the *Central Control*.

1 Introduction

Le *Central Control* (souvent abrégé CC dans la suite) est un composant logiciel qui a été conçu pour être le noyau d'un environnement pour le calcul scientifique. Le CC communique avec des serveurs qui peuvent être des systèmes de calcul formel généraux (comme Maple, Mathematica) ou des systèmes spécialisés, avec des outils de visualisation, ou avec des systèmes numériques (comme Matlab), ou encore avec des interfaces utilisateurs. Le principal rôle du CC est de fournir une vue commune et unifiée des différentes données manipulées par ses serveurs et des opérations qu'ils implémentent. Un résultat produit par un outil peut être envoyé à un autre après d'éventuelles transformations.

Lorsque nous avons conçu le *Central Control*, nous avons gardé à l'esprit deux buts : permettre l'ajout d'un nouveau serveur aussi simple que possible et une expérimentation aussi facile que possible.

La version courante du *Central Control* est un interprète *Scheme*, étendu avec de nouvelles opérations primitives et des types spéciaux. Seulement une petite partie du CC est en fait écrite en C (comme une extension réelle de l'interprète). La plus grande partie est directement écrite en *Scheme* et peut être modifiée facilement et dynamiquement. Les différents concepts se regroupent dans une bibliothèque *Scheme* qui permet d'effectuer diverses tâches de haut niveau, telles que les calculs distribués ou le choix d'un meilleur serveur.

2 Description du *Central Control*

Comme un *environnement pour le calcul distribué*, l'accent est mis sur une structure où les interactions entre les outils sont limitées en un certain sens : nous supposons que le temps de communication est négligeable par rapport aux temps de calculs, et que les outils sont largement indépendants (ils exécutent leur tâches sans l'aide d'autres outils). Le coût de passage d'une donnée par le CC reste alors linéaire en la taille de cette donnée. Typiquement, le CC peut être utilisé comme un gestionnaire de communication pour fournir un accès à plusieurs systèmes via une interface utilisateur (voir la figure 1), ou alors, pour fournir des services externes à une application qui a besoin des capacités de plusieurs systèmes algébriques. Un autre avantage d'un outil comme le *Central Control* est

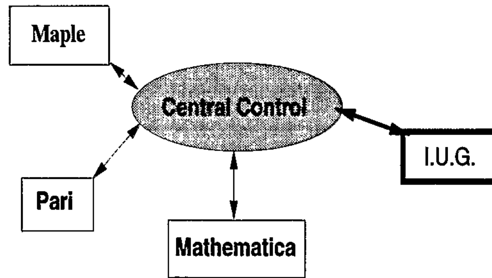

FIG. 1 – *Exemple de réseau*

la possibilité de réutiliser des bibliothèques déjà existantes avec peu de programmation supplémentaire. Le protocole de communication actuel est ASAP ([DGS94]). Il est prévu dans un futur proche de le remplacer par un protocole standard pour le calcul symbolique, OpenMath [Ope96].

Quelques systèmes et architectures logicielles ont été proposées depuis ces dernières années pour supporter un tel environnement, tels que par exemple POLYLITH [Pur94], *CaminoReal* [ABMW88], CAS/π [Kaj92] et SUI [DW90]. Mais notre approche est très différente car aucun de ceux-ci n'est réellement programmable dans le sens du *Central Control* (à travers un langage de programmation). SUI se focalise sur l'interface et n'est pas programmable au niveau de l'utilisateur. CAS/π et POLYLITH ont un "bus logiciel" qui rend l'addition de nouvelles fonctionnalités (ou l'accès à de nouvelles opérations à l'intérieur des serveurs) plus difficile et pas aussi dynamique que le CC.

2.1 Les fonctionnalités du CC

Nous avons choisi de fournir les principales fonctionnalités à travers des bibliothèques *Scheme*. Ainsi seules quelques primitives de base seront disponibles au niveau de l'utilisateur, la plupart essentiellement pour manipuler les serveurs. Par exemple, les fonctions (`server-create-local` *cmd*) et (`server-create-remote` *cmd machine*) lancent un serveur en utilisant la commande *cmd* respectivement sur une machine locale ou éloignée. Ces fonctions retournent un objet avec un nouveau type (*Scheme*) de base *server*.

2.1.1 Création et connexion

Nous distinguons deux types de serveurs : ceux que nous devons lancer nous-même et ceux qui existent déjà. Le premier type peut être obtenu via la fonction `server-create` et les autres via la fonction `server-connect`. Elles retournent un objet de type *server*. Ces deux fonctions sont écrites en utilisant des fonctions de base qui dépendent de la connexion (locale ou distante). Un serveur est terminé par la fonction `server-terminate`.

2.1.2 Calcul avec les serveurs

La fonction `server-compute` est utilisée pour envoyer des requêtes de calcul aux serveurs. La commande (`server-compute` *serv term*) où *serv* est un serveur et *term* un objet *Scheme* représentant la requête, envoie *term* à *serv* et retourne la valeur du résultat.

Dans l'exemple ci-dessous, nous créons un serveur PARI (un système de calcul formel spécialisé dans les calculs en théorie des nombres) qui est lancé sur la machine `kama.inria.fr` et affecté à un nouvel objet de type *server*.

```
> (define pari (server-create "kama.inria.fr" "pariserver"))
#<unspecified>
> pari
#<server>
```

Ce serveur appartient au service PARI. Un service est une abstraction commune pour les serveurs : il contient les règles de traduction pour les requêtes et résultats (voir plus loin). Nous pouvons maintenant envoyer une requête de calcul des 30 premiers nombres premiers :

```
> (server-compute pari '("primes" 30))
("seq" 2 3 5 7 11 13 17 19 23 29 31 37 41 43 47 53 59 61 67 71
 73 79 83 89 97 101 103 107 109 113)
```

2.1.3 Promesses

La fonction `server-compute` bloque le *Central Control* tant que le calcul n'est pas terminé dans le serveur. La fonction `server-batch` peut être utilisée pour envoyer une requête de manière "paresseuse" : la requête retourne immédiatement une *promesse* et l'utilisateur peut continuer son travail pendant que le serveur est en train de calculer.

Une promesse est un objet Scheme transformable, associée à un calcul en cours d'exécution dans un serveur donné. Il existe des primitives agissant sur les promesses, pour retrouver le serveur associé, pour vérifier si le calcul est terminé ou pour obtenir sa valeur lorsqu'elle est disponible. Quand la valeur d'une promesse est disponible, nous disons que

la promesse est réalisée. La fonction promise-ready? peut être utilisée pour tester si une promesse est réalisée.

La fonction Scheme first-one envoie simultanément la même requête de calcul *req* à deux serveurs différents s_1 et s_2. Dès qu'un des serveurs a terminé, l'autre est interrompu et le résultat est retourné. On effectue ainsi un calcul concurrent.

```
(define (first-one s1 s2 req)
  (let* ((p1 (server-batch s1 req)) (p2 (server-batch s2 req))
         (pr (wait-for p1 p2)))
    (server-interrupt s2)
    (server-interrupt s1)
    (promise-value pr)))
```

La primitive server-interrupt interrompt un serveur et promise-value extrait la valeur d'une promesse. wait-for prend des promesses comme arguments et retourne la première qui est réalisée.

Si nous incluons une promesse dans une requête de calcul, ou dans l'argument de server-compute, le CC se bloque jusqu'à ce que la promesse soit réalisée. Si nous incluons une promesse dans un appel server-batch, la fonction retourne immédiatement une autre promesse et la requête est mise en queue pour être envoyée quand la promesse est disponible.

2.1.4 "Handlers"

Comme les résultats d'un calcul peuvent être de très grande taille, il est souhaitable d'éviter systématiquement leur transmission vers le CC. Le CC peut associer un "*handler*" pour un résultat qui sera manipulé comme tout autre objet. Seule la requête d'une opération spécifique sur ce "handler" peut causer la transmission effective du résultat. Dans la plupart des cas, nous éviterons la transmission de la donnée associée au "handler", comme par exemple lorsqu'une requête (fonction du "handler") est envoyée vers le serveur qui possède la valeur de ce "handler". Cette notion de communication paresseuse réduit bien évidemment le temps de communication, mais aussi celui nécessaire pour la transformation des objets mathématiques dans la forme utilisée par le protocole d'intercommunication. L'espace mémoire alors utilisé à l'intérieur du CC est réduit puisque un seul objet de type pointeur est stocké.

Les fonctions handle-compute et handle-value permettent respectivement de demander à ce que la valeur de la requête de calcul soit stockée dans le serveur, et d'obtenir la valeur d'un "handler".

2.1.5 Règles de traduction

La philosophie du CC est de laisser le programmeur aussi libre que possible pour implémenter un serveur. Nous ne voulons pas imposer un standard pour l'encodage de tous les objets mathématiques. Donc nous avons besoin de fournir un mécanisme pour traduire la représentation utilisée par un serveur vers un autre. Cela signifie que pour n services, nous pouvons spécifier $n(n-1)$ traductions (une traduction dans les deux sens pour chaque paire de serveurs possibles). L'utilisateur peut également choisir une représentation unique des termes à l'intérieur du CC, et ainsi seules $2n$ traductions (du serveur vers cette

représentation, et inversement) sont alors nécessaires. Un ensemble de règles de traduction peut être associé à chaque service. En réalité, il y en a un pour l'envoi et un autre pour le retour. Chaque règle est constituée de modèles et utilisée par "pattern-matching". Une règle est une liste de deux éléments, un *modèle* et un *corps*. Un modèle contient des variables (ou des symboles) et un corps est une expression Scheme. Voici deux exemples de règles qui pourraient être utilisées pour traduire des termes envoyés vers un serveur *Maple*:

```
( ("factor" x) (if (integer? x) '("ifactor" ,x) '("factor"  ,x)) )
( ("integrate" . x) '("int" ,@x) )
```

Ce mécanisme permet l'utilisation d'une représentation commune dans le CC et manipule la sémantique de chaque serveur. Les ensembles de règles peuvent être structurés en utilisant une notion d'héritage.

La fonction `make-rule-set` appliquée à une liste de règles retourne tout simplement un ensemble de règles. Les arguments supplémentaires sont des ensembles de règles précédemment définis et dont le nouvel ensemble hérite:

```
(define pari-send-rule
  (make-rule-set '(   ((,integer? x) '("C" ,(number->string x)))  )
    standard-send-rule) )
```

L'ensemble `pari-send-rule` définit une règle spéciale pour traduire un entier *Scheme* en précision arbitraire en une forme spéciale utilisable par un serveur *Pari*.

2.1.6 Conditions exceptionnelles

Ils existent des conditions exceptionnelles qui doivent être transmises immédiatement à un serveur même si ce dernier est en train de calculer; par exemple les requêtes pour interrompre le calcul courant, ou pour demander des renseignements sur l'état du serveur (place mémoire occupée, temps CPU utilisé). `(server-interrupt` *server*`)` interrompt le calcul de *server*. `(server-busy?` *server*`)` retourne vrai si *server* est en train de calculer et faux sinon. `(server-status` *server*`)` retourne des informations qui dépendent du service. Cette commande est aussi utilisé pour obtenir les ressources utilisées par le serveur.

La fonction `server-terminate` utilise le même mécanisme pour être capable de terminer un serveur sans attendre la fin d'un calcul.

2.2 Une interface utilisateur

Pour créer une interface au CC avec une entrée classique des expressions mathématiques et une sortie en deux dimensions, nous avons utilisé plusieurs parties du programme JACAL [Jaf94] (un petit système de calcul formel écrit en *Scheme* par Aubrey Jaffer).

`ck(1)? maple := server(Maple,kama);`
`ck(1): <SERVER:1,ON:kama,SERVICE:Maple,STATUS:idle>`
`ck(2)? p := x^3 + 10*x - 3*x^3 + 20*x + x^2 - 2*x + 4*x^2 - 15;`
`ck(2):` $x^3 + 10 * x - 3 * x^3 + 20 * x + x^2 - 2 * x + 4 * x^2 - 15$
`ck(3)? q := compute(maple,p);`
`ck(3):` $-2x^3 + 28x + 5x^2 - 15$

Cette interface utilise un analyseur syntaxique dérivé de CGOL [Pra76] similaire à celui de *Macsyma* et peut ainsi facilement être étendu pour inclure de nouveaux opérateurs et constructions. := est la notation pour les affectations. Le symbole $ placé en fin de commande évite l'affichage du résultat. La valeur d'un objet de type *server* retourne plusieurs informations sur le serveur.

2.3 Implémentation

Le *Central Control* est une extension d'un interprète *Scheme* (*scm* pour l'instant). Elle est composée de 1200 lignes de C (en plus des 14000 de *scm*), de 1300 lignes de *Scheme* pour les primitives de base, et de 2000 autres lignes pour l'interface. L'exécutable est disponible sur Sparc (SunOS4) et DEC OSF/1. Sa taille en mémoire est de 700K sur Sparc dont 500K de texte partagé, ce qui correspond à la taille d'un noyau *Maple* au démarrage.

3 Une bibliothèque distribuée pour la géométrie algébrique

Quelque soit la stratégie employée, les opérations de la géométrie algébrique nécessitent un grand nombre de fonctionnalités disponibles dans certains systèmes qui, bien que complémentaires, sont très différentes sur un plan logiciel : calcul formel, géométrie algébrique, visualisation et algèbre commutative.

Or, écrire un programme répondant à toutes ces attentes serait complexe, surtout si nous voulons appliquer les principes de base du génie logiciel : modularité, extensibilité et maintenance. L'ajout de nouveaux outils demanderait la modification du travail existant, et l'efficacité de ce type d'approche dans une étape précise telle que le calcul d'une base de Groebner par exemple, serait certainement plus faible en regard d'un système spécialisé pour une telle tâche.

Notre approche consiste à définir et à implémenter les algorithmes distribués associés à de telles opérations dans l'environnement logiciel décrit précédemment lequel peut mettre à notre disposition les systèmes les plus adaptés et les plus performants pour de telles tâches. C'est cette bibliothèque, appelée *AlGeom*, que nous nous proposons de décrire dans cette section.

3.1 *AlGeom* : une bibliothèque Scheme sous le *Central Control*

AlGeom est une bibliothèque de fonctions *Scheme* qui fournit à un utilisateur du *Central Control* des opérations standards de la géométrie algébrique élémentaire. La différence par rapport aux implémentations courantes telles que celles des systèmes *Reduce* [Hea93], *Macaulay* [BS86] ou autres, provient essentiellement du fait que les algorithmes sont des algorithmes distribués utilisant un réseau de systèmes, chacun étant appropriés pour un calcul donné. Nous mettons donc à la disposition de l'utilisateur le meilleur outil pour chacun de ses sous-problèmes.

AlGeom contient dans sa première version les fonctionnalités pour définir des anneaux et des idéaux, pour calculer des bases de Groebner, pour calculer des sommes, des produits, des intersections, des quotients, des saturés ou des éliminés d'idéaux, pour tester si un idéal est premier, radical ou primaire, et pour calculer une décomposition première ou primaire d'un idéal donné. *AlGeom* utilise un ensemble de trois systèmes, *Maple*, *Macaulay*

et *GB* [Fau94]. Comme le CC est modifiable dynamiquement, et puisque cette bibliothèque est programmée de telle manière que les serveurs de calculs soient des paramètres, il est facile de remplacer un serveur par un autre plus approprié, ou bien de rajouter de nouveaux calculateurs. Dans ce dernier cas, il est nécessaire de modifier l'implémentation des algorithmes afin d'utiliser les nouvelles fonctionnalités du nouveau système pour obtenir un gain de temps ou d'espace mémoire.

3.2 Algorithmes dans *AlGeom*

Nous allons dans cette sous-section donner un bref aperçu de deux algorithmes implémentés dans *AlGeom*: l'élimination de variables et l'intersection d'idéaux. La base mathématique est extraite de [CLO92] et des détails sur l'implémentation sont donnés dans [Sau95].

3.2.1 Opérations de base

Comme aucun calcul n'est effectué dans le *Central Control*, chaque opération, chaque simplification doit être effectuée par un serveur approprié. Ainsi, nous appelons "opérations de base", les opérations élémentaires sur les polynômes ou ensembles de polynômes, lesquelles sont réalisées par un unique système. Dans la plupart des cas, ces fonctions se composent que de quelques lignes de code. Comme exemples, nous pouvons citer la multiplication, l'addition, le PGCD de deux polynômes, le degré, la factorisation, le calcul de base de Groebner etc ... Regardons quelques exemples en détails.

```
(define :gbasis (lambda args (server-compute gb '("gbasis" ,@args))))
```

Ce premier exemple spécifie que le calcul d'une base de Groebner s'effectue par l'intermédiaire du système *GB*. Cette fonction retourne une base de Groebner minimale réduite par rapport à l'ordre donné en argument.

```
(define (:gcd lpol)
  (cond ((or (not (list? lpol)) (not (string=? "bracket" (car lpol))))
         (:error ":gcd" ":" "bad argument"))
        ((= (length lpol) 2) (cadr lpol))
        (else (let ((strexp (:build-gcd (cdr lpol))))
                (server-compute maple strexp)))))
```

Cette fonction `:gcd` permet de calculer le PGCD d'une liste de polynômes via le système *Maple*. La procédure auxiliaire `:build-gcd` construit la requête de calcul en fonction de l'opérateur binaire (disponible sous *Maple*) GCD.

3.2.2 Une méthode d'élimination

Rappelons le théorème qui permet de calculer une élimination.

Théorème 1 *Soient $I \subset k[x_1, ..., x_n]$ et G une base de Groebner de I en respectant l'ordre lexicographique $x_1 > x_2 > ... > x_n$. Alors, pour tout $0 \leq k \leq n$, l'ensemble $G_k = G \cap k[x_{k+1}, ..., x_n]$ est une base de Groebner du k-ième idéal éliminé I_k.*

La commande liée à l'interface CK du *Central Control* permettant d'accéder à cette fonctionnalité est *eliminate(lpol, lvar)* avec *lpol* une liste de polynômes représentant le système d'équations à résoudre, et *lvar*, la liste des variables à éliminer. Le programme *Scheme* associé est :

```
(define (:eliminate lpol lvar)
  (let* ((res ())(newl (:elim-order lvar))(base (:gbasis lpol newl "lex")))
     (begin (for-each (lambda (i)
                        (if (not (:member? (cdr lvar) (cdr (:indets i))))
                            (set! res (cons i res))))
              (cdr base))
             (if (null? res)
                 '("bracket" 1)
              (cons "bracket" (reverse res))))))
```

où la fonction `:elim-order` réordonne la liste de telle manière que les variables à éliminer soient supérieures aux autres. Cette manipulation s'effectue grâce à *Maple*. Par exemple supposons que la liste des variables soit $[x, y, z, t]$ et que nous désirons éliminer les variables y, t. Alors la variable *newl* sera $[y, t, x, z]$. Ensuite nous appliquons le théorème 1, c'est à dire que nous calculons une base de Groebner lexicographique avec la nouvelle liste de variables par le système *GB*. Enfin, par *Maple*, nous examinons chaque générateur : s'il n'est pas fonction des variables à éliminer alors c'est un générateur de l'idéal éliminé.

3.2.3 Intersection de deux idéaux

Exemple : I un idéal dans $k[x, y]$, généré par le polynôme $f = (x+y)^4(x^2+y)^2(x-5y)$, et J un idéal généré par $g = (x + y)(x^2 + y)^3(x + 3y)$. Il est facile de voir que :

$$I \cap J = < (x + y)^4(x^2 + y)^3(x - 5y)(x + 3y) >$$

Ce calcul est facile précisément parce que les idéaux principaux f et g sont donnés sous forme factorisée (produit de polynômes irréductibles). Les algorithmes implémentés doivent donc être assez puissants pour traiter le cas où les idéaux ne sont pas principaux. Une astuce consiste à se ramener au calcul d'une intersection avec un sous-anneau (procédé d'élimination).

Théorème 2 *Soient I et J 2 idéaux de $k[x_1, ..., x_n]$. Alors $I \cap J = (tI + (1 - t)J) \cap k[x_1, ..., x_n]$.*

Nous avons choisi d'implémenter ce théorème, lequel consiste à utiliser les bases de Groebner. La commande CK est *inter*(id_1, id_2) et correspond au programme *Scheme* suivant :

```
(define (:inter id1 id2)
  (cond ((and (= (length id1) 2) (= (length id2) 2))
         `("bracket" ,(:lcm (cadr id1) (cadr id2))))
        (else (:inter-r id1 id2))))

(define (:inter-r id1 id2)
 (let ((freevar (car :list-of-free-var))
```

```
(:eliminate-r
  (:sum-r (cons "bracket" (:*-generators '(,freevar) (cdr id1)))
          (cons "bracket" (:*-generators '(("-" 1 ,freevar)) (cdr id2))))
  '("bracket" ,freevar)))
```

où `:sum-r` effectue la somme de deux idéaux, `:*-generators` correspond au produit de deux idéaux (via *Maple*), et `:list-of-free-var` est la liste des lettres minuscules et majuscules de l'alphabet moins les variables d'anneau.

3.2.4 Les prédicats

Certains prédicats sont disponibles dans *AlGeom*. Voici la commande CK des principaux :

- `principal?` retourne vrai si l'idéal est un idéal principal : pour cela, nous calculons par le système *GB* une base de Groebner minimale réduite pour l'ordre "revlex", et si la base ainsi trouvée ne comporte qu'un seul générateur, alors l'idéal donné est un idéal principal,
- `trivial_ideal?` retourne vrai si l'idéal donné est l'idéal trivial ($= k[x_1, ..., x_n]$) : pour cela, nous calculons une base de Groebner pour l'ordre "revlex" par *GB*, et nous regardons si l'égalité à une constante non nulle est vrai,
- `homo?` retourne vrai si l'idéal donné est un idéal homogène : nous vérifions pour cela l'égalité entre le plus haut et le plus bas degré du polynôme via les commandes `degree` et `ldegree` de *Maple*,
- `equal_id?` retourne vrai si deux idéaux sont égaux : pour cela nous comparons, par l'intermédiaire du système *GB*, leurs bases de Groebner respectives pour l'ordre "revlex".

4 Conclusion

La qualité principale du *Central Control* est certainement sa souplesse d'utilisation et d'extension. La taille du processus associé au CC reste raisonnable (elle est équivalente à celle d'un gros *shell*). Nous avons montré, avec la décomposition primaire d'idéaux polynomiaux [Sau95], que le *Central Control* peut être un outil facilement utilisable pour la réalisation de diverses expérimentations pour le calcul distribué et pour la coopération entre systèmes. Quelques améliorations peuvent néanmoins être envisagées en particulier pour la communication de grosses données. Pour le moment, ces communications sont coûteuses, notre protocole n'étant pas optimisé.

Notre approche semble prometteuse pour la construction d'un puissant environnement de calcul à partir de systèmes généraux et de serveurs spécialisés. Nous avons besoin d'autres expériences et d'autres applications, et nous envisageons une large utilisation de cet outil.

Remerciements

Ce travail a été partiellement financé par le projet ESPRIT PoSSo (6846).

Références

[ABMW88] Arnon (Dennis), Beach (Richard), McIsaac (Kevin) et Waldspurger (Carl). – CAMINOREAL: an Interactive Mathematical Notebook. *In : Proc. of EP'88 International Conference on Electronic Publishing, Document Manipulation and Typography.* – Nice, France, avril 1988.

[BS86] Bayer (D) et Stillman (M.). – The design of Macaulay : A system for computing in algebraic geometry and commutative algebra. *In : Proc. of 1986 Symposium on symbolic and algebraic manipulation.* pp. 157–162. – ACM.

[CLO92] Cox (David), Little (John) et O'Shea (Donal). – *Ideals, Varieties and Algorithms : An Introduction to Computational Algebraic Geometry and Commutative Algebra.* – Springer-Verlag, 1992.

[DGS94] Dalmas (Stéphane), Gaëtano (Marc) et Sausse (Alain). – *ASAP : a protocol for symbolic computation systems.* – Rapport Technique n°162, INRIA, mars 1994.

[DW90] Doleh (Y.) et Wang (P.S.). – SUI : A System Independant User Interface for an Integrated Scientific Computing Environment. *In : Proc. of ISSAC'90*, pp. 88–94.

[Fau94] Faugère (Jean-Charles). – *Résolution des Systèmes d'Équations Algébriques.* – Thèse de PhD, Université Paris 6, 1994.

[Hea93] Hearn (Anthony C.). – *REDUCE : User's Manual.* – RAND Publication CP178, octobre 1993.

[Jaf94] Jaffer (Aubrey). – JACAL home page on the World Wide Web. – `http://www.inf-gr.htw-zittau.de/~wagenkn/Schemedoku/`, avril 1994. Version 1a5.

[Kaj92] Kajler (Norbert). – CAS/PI: a Portable and Extensible Interface for Computer Algebra Systems. *In : Proc. of ISSAC'92.* pp. 376–386. – Berkeley, USA, juillet 1992.

[Ope96] Draft OpenMath home page on the World Wide Web. – `http://www.rrz.uni-koeln.de/themen/Computealgebra/OpenMath/`, mai 1996.

[Pra76] Pratt (V. R.). – Top Down Operator Precendence. *In : Proc. of the 1973 ACM Symposium on Principles of Programming Languages.* pp. 41–51. – ACM Press.

[Pur94] Purtillo (James M.). – The POLYLITH Software Bus. *ACM Transactions on Programming Languages and Systems*, vol. 16, janvier 1994, pp. 151–174.

[Sau95] Sausse (Alain). – *Architecture Logicielle Distribuée pour le Calcul Formel. Application à la Décomposition Primaire d'Idéaux.* – Thèse de PhD, Université de Nice-Sophia Antipolis, décembre 1995.