UNIVERSITE DE NICE SOPHIA-ANTIPOLIS

De la modélisation littérale à la simulation numérique certifiée

Mémoire présenté et soutenu le 22 juin 2012 par

Yves Papegay

pour obtenir le diplôme d'

Habilitation à Diriger des Recherches

Sciences et Technologies de l'Information et de la Communication

Devant le jury composé de :

Laurent Farenc	
Claude Gomez	(Rapporteur)
Luc Jaulin	(Rapporteur)
Michel Rueher	
Stephen Watt	(Rapporteur)

Table des matières

Introduction **1**

1 Bien modéliser pour mieux simuler **7**
1.1 À propos de modèles 8
1.1.1 Orientation des équations 9
1.1.2 Ordre des équations 11
1.2 Modèles de l'ingénieur 12
1.3 De la modélisation à la simulation 14
1.4 Quelques contributions 16
1.4.1 En mécanique 16
1.4.2 En électronique 17
1.4.3 En optique 17
1.4.4 En aéronautique 18

2 Efficacité, précision et exactitude **21**
2.1 Calcul symbolique versus calcul numérique 21
2.1.1 Apports et limitations du calcul symbolique 22
2.1.2 Apports et limitations du calcul numérique 24
2.2 Algorithmes symboliques–numériques 25
2.3 Analyse par intervalles 26
2.4 Quelques contributions : applications en robotique 28

3 Quelques contributions logicielles **31**
3.1 Représenter les expressions 32
3.2 Communiquer 33
3.3 Générer du code C 33
3.4 Calculer 35

4 Un environnement logiciel pour la modélisation et la simulation **37**
4.1 Présentation générale 37
4.1.1 Motivations 37
4.1.2 Besoins 38
4.1.3 Existant 39

4.1.4 Architecture 41
4.2 Conception des modèles 42
4.2.1 Langage Simple de Modélisation - LSM 43
4.3 Analyse de modèles 48
4.3.1 Validation sémantique 48
4.3.2 Variables, dépendances et complétude 51
4.3.3 Calculs de grandeurs et d'unités 55
4.4 Compilation 57
4.4.1 Ordonnancement 58
4.4.2 Typage dynamique 58
4.4.3 Génération de code 60
4.5 Réalisations 61

Perspectives **63**

Références **65**

Publications **67**

Curriculum Vitae **73**

À Jacques et à ma maman, à qui j'aurais tant aimé faire lire ce mémoire ...

À Vincent, Fantine, qui comprendront peut-être un jour en le lisant combien ils m'ont aidé à l'écrire ...

À Laure qui n'a pas besoin de le lire pour savoir ...

À Luc, Claude, Stephen et Laurent qui l'ont lu, annoté et corrigé ...

À Jean-Pierre et à Gérard sans qui je ne l'aurais probablement pas écrit.

Introduction

J'ai fait partie de la première promotion du DEA "Mathématique-Informatique" de l'Université de Nice. Ce Diplôme d'Etudes Approfondies était alors dédié au calcul formel et possédait trois composantes fortes. La première composante était algébrique et l'on y étudiait les bases standards (ou de Gröbner) et notamment l'algorithme de Buchberger. Dans une deuxième, les mécanismes calculatoires et les calculs de complexité étaient passés au crible, notamment en ce qui concerne l'évaluation des formes linéaires. On se consacrait dans la troisième à des implémentations expérimentales ; on programmait en Macsyma puis en Scratchpad et en Maple à travers de lourds protocoles à la fois informatiques et administratifs qui connectaient quelques terminaux alphanumériques niçois au serveur providentiel du Centre InterUniversitaire de Calcul de Grenoble. Je partageai ainsi l'enthousiasme naissant d'une communauté de mathématiciens-algébristes pour des techniques et des outils qui leur permettraient de manipuler et de calculer effectivement avec leurs objets favoris, sans restreindre ni leur diversité ni leur abstraction.

C'est aussi au cours de cette année que je rencontrai le monde industriel en réalisant, au sein du bureau d'étude de l'Aerospatiale Cannes, le projet individuel qui avait une large place dans ce diplôme. Cette rencontre, avec celles de jeunes ingénieurs dynamiques, enthousiastes et motivés, Christophe Garnier et Pascal Rideau, détermineraient le choix de mon sujet de thèse, et donneraient dorénavant à la plupart de mes recherches, une certaine orientation vers les applications industrielles.

Dans le sillage des Professeurs André Galligo et Jacques Morgenstern, mes années de thèse sont indissociables du démarrage de l'équipe-projet SAFIR – pour "Systèmes Algébriques et Formels pour l'Industrie et la Recherche" –, équipe-projet commune entre le laboratoire de Mathématique de l'Université de Nice - Sophia Antipolis et l'Unité de Recherche de Sophia Antipolis de l'Institut National de Recherche en Informatique et en Automatique (INRIA). Se retrouvaient dans cette équipe-projet les trois composantes du DEA transcendées en thématiques de recherches, la plus représentée étant la résolution de systèmes algébriques et plus généralement la géométrie algébrique effective. Celle-ci restera longtemps – et reste encore ? – une thématique primordiale dans la communauté académique française en calcul formel. Une composante informatique, née des expériences précédentes, s'attaquait aux problèmes spécifiques d'implémentation efficace de structures de données et d'algorithmes pour le calcul symbolique en cherchant notamment à exprimer ces algorithmes dans toute

leur généricité. L'étude des mécanismes calculatoires poussait à considérer les codes de calculs numériques comme des objets que l'on pouvait symboliquement analyser, manipuler et transformer et sur lesquels on pouvait calculer. Ceci offrait de belles perspectives et introduirait plus tard les travaux sur la différentiation automatique. Enfin la présence du "I" dans l'acronyme SAFIR – pour "Industrie" – n'était pas fortuite mais bien représentative d'une orientation de l'équipe vers les applications concrètes et les utilisateurs potentiels des techniques étudiées.

Dans cet environnement riche, j'étudiais pendant ma thèse comment générer automatiquement les équations gouvernant la dynamique de systèmes mécaniques rigides poly-articulés, à partir d'une description structurelle de ces systèmes. Je regardais aussi si l'algèbre pouvait se substituer à l'intuition de l'ingénieur mécanicien pour traiter automatiquement les cas durs où ces systèmes étaient cinématiquement bouclés. L'ensemble de ces travaux était toujours réalisé en étroite collaboration avec le bureau d'étude de l'Aerospatiale à Cannes.

Le début des années 90 a vu la communauté de calcul formel s'intéresser à la résolution de systèmes d'équations polynomiales issus de la robotique, et notamment au modèle géométrique direct de la plateforme de Stewart[Laz92]. Ce fut l'occasion de mes premiers contacts avec Jean-Pierre Merlet qui me fera découvrir la diversité des problèmes liés à la modélisation des robots parallèles, et me fera croiser le chemin du Dr Hirohisa Hirukawa dans l'équipe duquel j'attaquai alors, en séjour post-doctoral au sein du prestigieux laboratoire de robotique ElectroTechnical Laboratory à Tsukuba au Japon, une carrière de chercheur. Cet épisode nippon fut très enrichissant, tant sur le plan scientifique que culturel et personnel. Je découvrais notamment que dans un certain référentiel, recherche et développement ne sont pas des activités qui s'opposent, mais bien complémentaires et mutuellement indispensables, comme le sont théorie et expérimentation dès que l'on souhaite faire fonctionner effectivement un robot.

J'avais hérité de ma formation universitaire une double compétence mathématique et informatique, et du contexte de collaboration industrielle de ma thèse un goût prononcé pour l'application concrète. De mes années passées dans l'équipe-projet SAFIR, j'avais acquis expérimentalement une solide connaissance des techniques et des systèmes de calcul formel existants, de leurs fonctionnalités, de leur puissance et de leurs limitations. En collaborant avec un bureau d'étude industriel, j'avais acquis une solide connaissance des besoins liés à la conception de systèmes de plus en plus complexes, et des contraintes opposables en terme de rapidité de prototypage, de performance de simulation, de précision et de justesse des résultats. J'avais alors la profonde conviction que la possibilité de calculer avec des symboles – en utilisant la puissance informatique – en se libérant provisoirement de la nécessité d'une évaluation numérique, devait, dans le contexte des méthodes d'ingénierie, permettre de grands progrès dans la modélisation des phénomènes physiques et par voie de conséquence dans la fiabilité, l'efficacité et la qualité de la simulation du comportement des systèmes complexes.

En me proposant de rejoindre les rangs de ses chercheurs, INRIA m'offrit la possibilité de concrétiser cette conviction, au fil des années, en un objectif, celui de concevoir et de

développer un environnement logiciel pour la modélisation et la simulation qui possède de solides fonctionnalités aussi bien formelles que numériques, et qui permette à l'ingénieur en charge de l'élaboration de modèles d'obtenir automatiquement in fine le code de simulation correspondant sans avoir eu à se soucier, ni des méthodes de résolution éventuellement mises en oeuvre, ni de contraintes informatiques d'implémentation.

De fait, depuis cette époque, une grande partie de mes activités de recherche est restée centrée sur l'application des techniques et des outils de manipulation symbolique à des problèmes de conception, de modélisation et de simulation. Les domaines d'applications ont été très variés. J'ai étudié des problèmes et obtenus des résultats en mécanique tout d'abord (mécanique des systèmes poly-articulés), puis en robotique (étude d'espaces de configurations pour la planification de mouvements d'assemblage), en optique ensuite (bilans de performances d'interféromètres embarqués), ainsi qu'en électronique (simulation de circuits à base de transistors) et enfin en aérodynamique (simulation de la dynamique du vol d'avions). A chaque fois, bien sûr, les problèmes théoriques étudiés sont différents, et mes travaux, toujours effectués dans un environnement applicatif industriel, sont guidés par la nécessité de fournir à ces problèmes une réponse concrète, le plus souvent à travers une implémentation logicielle.

La richesse de l'environnement scientifique du Centre de Recherche Sophia Méditerranée et les caprices d'un destin cruel qui a mis fin prématurément à l'équipe-projet SAFIR m'ont conduit à être aussi confronté avec les notions qui interviennent dans la qualité numérique des codes de calculs, et plus précisément de m'intéresser aux calculs de sensibilité et aux calculs de propagation des incertitudes.

Successivement chercheur dans les équipes-projets SAGA (pour "Systèmes Algébriques, Géométrie et Applications") et COPRIN (pour "Contraintes, OPtimisation, Résolution par INtervalles"), j'ai consacré une partie de mon activité à l'étude de propriétés des robots parallèles et des robots à câbles, qui étaient le domaine applicatif privilégié ces deux équipes-projets. En parallèle de mes activités dans le domaine de la modélisation-simulation, je me suis aussi intéressé aux méthodes et aux outils de l'analyse par intervalles et de la programmation par contrainte, coeur de métier de l'équipe-projet COPRIN, et ces deux intérêts ont convergé vers la prise en compte des incertitudes et les calculs de domaines de validité dans les problèmes de modélisation industrielle.

Plan du mémoire

Ce mémoire, dont l'objet est de présenter une synthèse de mes travaux de recherche et de les mettre en perspective avec l'objectif général de la conception et du développement d'un environnement logiciel pour la modélisation et la simulation est composé de quatre chapitres, les trois premiers ayant pour objets mes travaux et le quatrième cet environnement.

Contributions

Les travaux présentés dans les trois premiers chapitres de ce mémoire sont développés selon trois axes thématiques. Le quatrième, plus technique, présente les contours et les fonctionnalités de cet environnement logiciel intégré pour la modélisation et la simulation, dont la conception a été le cadre général et le fil conducteur de l'essentiel de mes travaux et dont une première version est sur le point d'être utilisée par Airbus.

Bien modéliser pour mieux simuler La finalité de la simulation est claire, puisqu'il s'agit d'apprécier le comportement d'un système complexe dans un certain environnement sans avoir à plonger le système dans cet environnement. Toutefois, une réflexion est nécessaire pour bien identifier et analyser toutes les étapes, qui, partant du système à étudier, conduisent à en établir un modèle physique tout d'abord, un modèle calculatoire par la suite pour aboutir à un programme de simulation. Ainsi les concepts de grandeurs, de variable, de données, de relations, d'équation, de formalisme, de modèle d'une part, les propriétés attachées à ces concepts et à leurs instances d'autres parts, méritent d'être formellement définis et établis. Ces définitions précises sont aussi la base nécessaire pour mettre en place des langages de description plus ou moins spécialisés des systèmes et des modèles. L'analyse des textes de ces langages fournira les informations nécessaires à la fabrication de programmes de simulation et c'est bien l'existence de tels langages qui permettra une automatisation de ce processus.

Dans ce cadre de réflexion théorique sur les objets et les langages de modélisation, j'ai notamment étudié en détail l'influence du choix de la méthode et du formalisme sur la possibilité d'exploiter un modèle avec des outils de calcul symbolique. Je me suis intéressé aussi à la notion d'orientation des équations dans les modèles et l'impact sur la production d'un modèle calculatoire à partir d'un modèle équationnel. Pour cela, des algorithmes originaux ont été développés, basés sur une représentation des dépendances entre les paramètres en termes de graphes. Plusieurs implémentations ont été réalisées, produisant de nombreux résultats applicatifs en mécanique, en électronique, en optique et en aéronautique.

Calculer efficacement, précisément et juste Les phénomènes de croissance parfois exponentielle des expressions intermédiaires est une limitation bien connue de certains algorithmes de calcul formels, d'autant que la taille du résultat final n'est pas forcément concernée. Quand on conserve la perspective de calculer des résultats numériques à partir des expressions formelles, ce phénomène peut devenir bloquant au niveau des performances mais aussi au niveau de la qualité numérique des résultats obtenus. Dans certains cas, une adaptation des méthodes symboliques utilisées est suffisante pour retrouver des résultats utilisables alors que dans d'autres cas, il est nécessaire d'abandonner ces méthodes pour d'autres méthodes fournissant des résultats certifiés.

Travaillant sur l'exploitation numérique des calculs menés symboliquement, j'ai mis en

évidence des exemples relativement simples en cinématique des mécanismes où les méthodes symboliques échouent par la taille des expressions intervenant dans les calculs et sur lesquels les méthodes numériques génériques fournissent des résultats erronés,

L'utilisation d'une arithmétique d'intervalles permet de certifier les calculs numériques effectués tout en tenant compte des incertitudes inhérentes à certains paramètres impliqués dans ces calculs. Un inconvénient du calcul par intervalles est le phénomène intrinsèque de surestimation du résultat et les méthodes développées dans le cadre de l'analyse par intervalles visent à minimiser les conséquences de cette surestimation, en adaptant les algorithmes numériques classiques à cette arithmétique différente.

Avec l'objectif de réaliser des simulations numériques certifiées efficaces de modèles symboliques, j'ai développé des adaptations aux intervalles d'algorithmes hybrides symboliques-numériques, et généralisé le principe d'un préconditionnement symbolique pour l'évaluation par intervalles. Ces méthodes ont été appliquées à la résolution de modèles de robots parallèles.

Générer un code de calcul dédié Lorsque la modélisation se fait dans un cadre industriel, pour des systèmes de grande taille ou avec des perspectives de simulation en temps-réel, et même si l'on ne rencontre pas les problèmes intrinsèques de performances ou de qualité logicielle décrits ci-dessus, il est fréquent de devoir laisser la responsabilité des calculs numériques à des langages conçus pour ces calculs comme le Fortran ou le C. Dans ces cas, la connaissance d'une description des modèles dans un langage symbolique de haut-niveau permet une génération complètement automatisable du code de simulation. Cette génération peut s'assimiler à la production de code cible d'un compilateur et comporte les mêmes étapes d'analyse syntaxique et sémantique de représentation intermédiaire du code et d'éventuelle optimisation de ce code. Les techniques en jeux pour écrire ce "compilateur symbolique" sont toutefois spécifiques de la nature du langage source.

Dans ce domaine mon travail s'est concrétisé par deux projets logiciels qui ont débouché sur des transferts industriels : une représentation sémantique formelle du langage C dans le système de calcul formel Mathematica, et un environnement de modélisation-simulation pour la production de code des simulateurs de vol Airbus.

Différentes contributions logicielles portant sur la manipulation symbolique d'expressions, la communication d'objets mathématiques et les calculs à base d'intervalles sont aussi présentées dans ce chapitre.

Un environnement logiciel pour la modélisation et la simulation

Ce dernier chapitre est fortement inspiré d'un document inédit de spécification de l'environnement logiciel développé depuis 2005 dans le cadre d'une collaboration avec Airbus dans lequel certains détails ont été omis pour des raisons de confidentialité, il présente les

actuels besoins industriels en terme de fonctionnalités d'un tel environnement et les pistes que nous avons suivies pour les satisfaire.

Une dernière section présente le prototype qui a été effectivement réalisé, évoque certaines difficultés de réalisation et les solutions développées pour les surmonter.

Conclusion

En guise de conclusion, on évoque des perspectives d'évolutions nécessaires pour les actuels outils de calculs formel pour devenir les environnements informatiques idéaux pour adresser les problèmes de modélisation/simulation évoqué tout au long de ce mémoire.

Chapitre 1

Bien modéliser pour mieux simuler

Lorsque l'on conçoit un nouvel objet, une nouvel équipement, une nouvelle technique, un nouveau système complexe, on le veut conforme à certaines *spécifications* qui décrivent ce qu'on attend de son comportement en termes de fonctionnalités, de résultats, ou de performances.

Pour être guidé pendant les phases de conceptions, et être aidé à prendre une décision à chaque fois qu'il se trouve face à un choix (forme, matériau, solution technique, algorithme, structure, topologie, etc.), le concepteur a besoin d'évaluer par avance le comportement de l'objet ou du système avant même que ne soit figée précisément sa définition.

Pour cela il va chercher à *simuler* ce comportement, par exemple en étudiant le comportement d'un prototype, ou d'une maquette de l'objet, c'est à dire d'un objet plus simple et plus facile à réaliser mais qui est censé avoir des fonctionnalités et des performances proches, ou tout au moins dont l'étude et la mesure doit permettre d'obtenir des informations sur l'objet en cours de conception. Cette approche de la simulation est expérimentale.

Une approche complémentaire, plus théorique, parfois concurrente, consiste à élaborer un *modèle*. Il s'agit d'assembler des concepts qui représentent, qui permettent de comprendre et de prédire le système que l'on conçoit, en exploitant les connaissances scientifiques, technologiques et techniques concernées, et en exploitant l'analogie entre ce système et des systèmes déjà existants, étudiés et bien connus.

Nous allons nous intéresser dans ce chapitre à cette deuxième approche, basée sur la modélisation, en détaillant notre vision de cette activité de *modélisation/simulation* qui, comme on vient de le voir, est au coeur de nombreux processus de conceptions, mais que l'on retrouve aussi avec la même problématique dans de nombreux autres composants de l'activité humaine, notamment scientifique et technologique.

1.1 À propos de modèles

Si l'on se réfère à la définition d'un dictionnaire de la langue française, on désigne par modèle "toute structure logique ou mathématique formalisée, utilisée pour rendre compte d'un ensemble de phénomènes qui, bien que n'ayant pas de lien de causalité univoque, possèdent entre eux certaines relations". Dans cette définition, les termes importants sont "rendre compte", "relations" et "formalisée". Ils font référence aux trois étapes décisionnelles de l'élaboration du modèle du comportement d'un système complexe :

1. identifier quels sont les phénomènes représentatifs de ce comportement dont on souhaite rendre compte, et leur attacher un certain nombre de *grandeurs* physiques qui en sont caractéristiques,
2. associer à ces grandeurs une ou plusieurs *théorie physique* connue – par théorie on entend un ensemble précisément identifié et cohérent de relations qui font intervenir des grandeurs pour traduire la connaissance scientifique que l'on a des phénomènes qui les font intervenir,
3. exprimer ces relations au moyen d'un formalisme précis qui permette d'associer les grandeurs physiques caractéristiques à des variables qui les représentent et de transcrire en termes d'équations et d'inéquations les relations physiques entre ces grandeurs.

Chacune de ces étapes correspond à des choix de la part de l'ingénieur, que l'on pourrait croire naturellement guidés par la nature physique du système et des grandeurs étudiés. Mais dans la plupart des cas, plusieurs approches scientifiques et plusieurs formalismes existent pour représenter le même système et étudier un même type de comportement.

Prenons un exemple : si l'on s'intéresse au modèle du comportement dynamique d'un mécanisme, on pourra choisir de considérer des grandeurs comme la masse, l'inertie, la position, et l'accélération de chacune des pièces du mécanisme et les considérer dans le cadre du principe fondamentale de la dynamique. Mais l'on pourra aussi préférer introduire les vitesses de ces composants et l'énergie cinétique du système pour utiliser le principe variationnel de Lagrange. Dans tous les cas, pour donner une existence concrète et mathématique au modèle, il faudra en plus choisir un *formalisme* pour représenter l'orientation des corps de notre mécanisme, et quel que soit le choix fait précédemment, on pourra utiliser des quaternions, des angles d'Euler ou de Bryant, voire des paramètres de Denavit-Hartenberg...

A travers leur influence fondamentale sur la manière d'écrire les équations du modèle physique, la méthode et le formalisme retenus conditionnent la facilité, voire la possibilité d'expliciter ces équations et de produire un modèle calculatoire, et ces choix sont donc décisifs quant à la possibilité d'obtenir in-fine un code de simulation avec de bonnes propriétés. Ils conditionnent la complexité du modèle, la fidélité et la précision des résultats qu'il permet de calculer ainsi que la qualité et les performances de ce code de simulation.

J'ai étudié en détail cette influence du choix de la méthode et du formalisme sur la possibilité d'exploiter un modèle avec des outils de calcul symbolique. L'exemple traité est celui de la génération des équations de la dynamique d'un système de corps rigides poly-articulés simple (un satellite transportant une antenne orientable). Cette étude constitue le deuxième chapitre de
[1] *Outils formels pour la modélisation en mécanique*. Thèse de Doctorat, Université de Nice Sophia-Antipolis, France, Novembre 1992.

Pour préciser le propos, on peut donner une définition précise d'un modèle tel qu'on vient de l'évoquer

Définition 1 (Modèle formel) *: Un* modèle formel *est caractérisé par la donnée d'un ensemble de paramètres,* $\{p_i\}_{i=1..m}$ *, d'un ensemble d'équations,* $\{f_j(p) = 0\}_{j=1..n}$ *, et d'un ensemble d'inéquations* $\{g_k(p) < 0\}_{k=1..r}$.

Dans cette définition rien ne restreint, a priori, la généralité des équations qui peuvent être algébriques, intégrales, ou différentielles. Toutefois, pour être exploité, ce modèle formel doit être associé à un modèle mathématique calculatoire qui explicite comment on peut calculer certaines de ces grandeurs en fonction d'autres et qui permet d'exécuter ces calculs. Les deux sections suivantes précisent les étapes nécessaires pour obtenir ce modèle calculatoire.

1.1.1 Orientation des équations

En décrivant des relations multi-latérales entre des grandeurs physiques, les équations du modèle formel tel qu'on l'a défini ci-dessus sont généralement implicites et non-orientées, tandis que les problèmes d'ingénierie pour lequel le modèle a été développé le sont intrinsèquement : parmi les paramètres du modèle, certains sont connus, d'autres sont à calculer.

Lors de la conception d'un système, se succèdent en alternant, des phases de synthèse et des phases d'analyse, au cours desquelles respectivement, on détermine des paramètres de conception, et on évalue leur influence sur le comportement du système :
– pendant les phases de synthèse, on définit quel est le comportement souhaité du système pour un environnement donné, et on recherche une architecture et des dimensions qui permettent au système d'avoir ce comportement.
– pendant les phases d'analyse, au contraire, l'architecture du système et ses caractéristiques sont fixées et, pour un environnement donné, on cherche à connaître le comportement du système en terme de performances.
Toutes ces phases, lorsque l'on s'intéresse au même système et au même type de comportement s'appuient sur le même modèle formel. Mais pour être mises en oeuvre, les équations (et les inéquations) du modèle sont explicitées et orientées différemment et les modèles obtenus sont caractérisés par des ensembles d'équations différentes bien qu'équivalentes.

Si l'on reprend l'exemple du comportement dynamique d'un mécanisme, le modèle formel exprimera la loi fondamentale de la dynamique en établissant, pour chaque corps, le lien entre efforts, masse et accélération avec la formulation classique annulant le bilan des forces : $F - (m.\gamma) = 0$

Selon les cas, on explicitera cette relation en écrivant au choix :
– lorsque l'on s'intéresse à tester la résistance du système, $F = m.\gamma$
– lorsque l'on veut analyser ses performances dynamiques, $\gamma = \frac{F}{m}$
– et si l'on souhaite dimensionner sa masse, $m = \frac{F}{\gamma}$

Là encore, on peut préciser cette notion par une définition.

Définition 2 (Modèle orienté) *Un* modèle orienté *est caractérisé*
– *par la donnée de trois ensembles de paramètres, les paramètres d'*entrée $\{e_i\}_{i=1..m_e}$ *, les paramètres de* sortie $\{s_i\}_{i=1..m_s}$ *, et les paramètres* locaux $\{l_i\}_{i=1..m_l}$
– *par la donnée de deux ensembles d'équations explicites* $\{s_j = \mathrm{fs}_j(e,s,l)\}_{j=1..n_s}$ *et* $\{l_j = \mathrm{fl}_j(e,s,l)\}_{j=1..n_l}$ *où e, (reps. s et l) est une notation abrégée pour* $(e_i)_{i=1..m_e, i\neq j}$
– *par la donnée éventuelle d'un ensemble d'inéquations* $\{g_k(e,s,l) < 0\}_{k=1..r}$

On passe ainsi du modèle formel au modèle orienté en partitionnant l'ensemble des paramètres en trois en fonction de leur nature et en orientant chaque équation $f_j(p) = 0$ en l'explicitant sous la forme $p_{i(j)} = \hat{f}_j(e,s,l)$ après avoir fait le choix d'un paramètre $p_{i(j)}$ parmi les paramètres locaux ou de sortie.

Il y a deux difficultés à surmonter pour réaliser effectivement ce passage dans la pratique :

1. Il n'est bien sûr pas toujours possible de résoudre l'équation $f(p_1, .., p_n) = 0$ par rapport à un quelconque des paramètres p_i qui y interviennent pour écrire formellement une équation équivalente $p_i = F(p_1, \ldots, p_{i-1}, p_{i+1}, \ldots, p_n)$. Il y a des cas où cette expression analytique n'existe pas et où l'on devra lui substituer un processus de résolution numérique, et il y a aussi des cas où il n'y a pas unicité de la solution symbolique et où l'on devra choisir. Enfin il y a des cas ou le calcul de cette solution est simplement trop coûteux en temps ou produirait un résultat trop important en taille des expressions pour être pratiquement utilisable.

2. Le choix du paramètre $p_{i(j)}$ pour le calcul duquel on va expliciter l'équation $f_j(p) = 0$, ne se fait pas simplement localement, en ne regardant que cette équation et en fonction de l'expression de l'équation. Il résulte de l'analyse globale de l'ensemble des équations du modèles et des paramètres qu'elles font intervenir, qui détermine les dépendances entre les paramètres et garantit, en imposant l'orientation des équations. la possibilité de mener à bien les calculs voulus.

Ces questions d'orientation de modèles ont été discutées précisément et plus particulièrement dans le rapport technique
[57] *ypama : un assistant à la modélisation, comment implémenter la désorientation des modèles.* Rapport d'étude, INRIA/Airbus France, Décembre 2003.

1.1.2 Ordre des équations

Le modèle orienté que l'on vient de voir n'est pas directement traduisible en un code de calcul, car il ne se préoccupe pas de l'ordonnancement du calcul.

Le modèle qui va pouvoir être traduit en un code numérique de simulation s'obtient à partir du modèle orienté évoqué ci-dessus en ordonnant les équations selon un ordre topologique respectant les dépendances des paramètres qu'expriment ces équations. On définit ainsi un troisième type de modèle que l'on obtient après avoir ordonné et orienté un modèle formel :

Définition 3 (Modèle calculatoire) *Un* modèle calculatoire *est caractérisé par*
– *la donnée de 3 ensembles de paramètres, connus ($\mathcal{E}$), à calculer ($\mathcal{S}$) et locaux ($\mathcal{L}$)*
– *d'une liste* ordonnée *d'équations* $(eq_j)_{j=1...n}$ *vérifiant*
  $eq_1 : q_1 = f_1(p)$ *où* p *représente les paramètres connus de* $\mathcal{E}$
  $eq_j : q_j = f_j(p, q_1, \ldots, q_{j-1})$ *pour tout* j,
  $\mathcal{S} \subset \{q_j\}_{j=1...n}$
– *et éventuellement d'un ensemble de contraintes* C *(sous la forme d'inéquations).*

Ainsi l'obtention d'un modèle calculatoire est étroitement liée à l'analyse et à la connaissance des dépendances entre les paramètres au travers des équations du modèle formel. qui permet de construire itérativement la liste d'équations en gérant les paramètres connus ou déja calculés.

Quelques problèmes intéressants se posent quant à l'obtention éventuelle d'un modèle calculatoire, avec des applications immédiates en ingénierie, particulièrement intéressantes si les modèles sont gros.

1. Etant donné un modèle orienté peut-on ordonner ses équations pour obtenir un modèle calculatoire ?
2. Etant donné un modèle formel et un partitionnement de ses paramètres en paramètres connus et à calculer peut-on trouver un modèle calculatoire avec ce partitionnement ?
3. Etant donné un modèle formel et un partitionnement $\{\mathcal{E}, \mathcal{S}, \mathcal{L}\}$ de ces paramètres tel qu'on ne peut pas trouver un modèle calculatoire avec ce partitionnement, existe-t-il un ensemble de paramètres de sorties ou locaux $\{p_j \in \mathcal{S} \cup \mathcal{L}\}_{|}$ tel que le partitionnement qui les considère comme connus, $\{\mathcal{E} \cup \{p_j\}, \mathcal{S} \setminus \{p_j\}_j, \mathcal{L} \setminus \{p_j\}_j\}$ permet de définir un modèle calculatoire ? et dans l'affirmative peut-on déterminer un ensemble minimal - et avec quel critère de minimalité - possédant cette propriété ?

Des algorithmes efficaces ont été développés pour attaquer ces problèmes, basés sur une représentation des dépendances entre les paramètres en termes de graphes.

Ces algorithmes ont été implémentés notamment dans les prototypes logiciels Lacapio et Ypama décrits dans les rapports techniques
[51] *Un logiciel d'aide à la conception et à l'évaluation des performances d'instruments optiques.* Rapport Technique, aerospatiale/INRIA, février 1995.
[56] *ypama : un assistant à la modélisation.* Rapport d'étude, INRIA/Airbus France, Janvier 2002.
Certains d'entre eux seront présentés dans la deuxième partie de ce mémoire.

1.2 Modèles de l'ingénieur

Les considérations théoriques précédentes n'entraînent pas explicitement de restriction ni sur les modèles ni sur la nature des équations et des variables qui y interviennent. Dans ce cadre général purement théorique et symbolique, on peut ainsi imaginer qu'une variable d'un modèle soit une fonction par exemple du temps, et que les équations qui la font intervenir soient des équations différentielles ou intégrales. On peut aussi rencontrer des équations aux dérivées partielles, des équations transcendantales, etc.

Mais si l'on a en tête la simulation concrète du comportement d'un système, une phase de résolution de ces équations est en général nécessaire pour obtenir un modèle calculatoire, et l'on sait que cette phase ne débouche souvent pas sur une expression littérale explicite unique : pour la plupart des équations différentielles, on utilisera des processus d'intégration numérique, pour la plupart des équations algébriques, on n'aura pas l'unicité des solutions, ni l'existence d'une expression symbolique associée, enfin, pour de nombreuses équations, on utilisera des processus itératifs numériques.

Pour pouvoir considérer aussi ces cas-là, on doit étendre la notion d'équation des modèles – qui était limitée jusque là à une égalité entre une variable et une expression formelle faisant intervenir d'autres variables – pour tenir compte des cas où les calculs s'appuient sur un processus calculatoire. Ainsi, on doit détailler dans la définition 3 la nature de ces fonctions $f_j(p, q)$ en précisant qu'il peut s'agir :
– d'expressions symboliques,
– de *programmes de calculs*,
– ou de tables numériques associées à des fonctions d'interpolation et d'extrapolation.
Dans tous les cas, on doit savoir évaluer numériquement une telle fonction pour n'importe quelle valeur des paramètres qui y interviennent, et si c'est la seule information dont on dispose, on parlera de *boîte noire*, mais il se peut que l'on ait des informations supplémentaires sur la nature du processus numérique, ou par exemple un moyen d'évaluer sa dérivée par rapport aux paramètres, on parlera alors de *boîte transparente*.

L'utilisation concrète de modèles dans le cadre de l'ingénierie nécessite aussi d'attacher aux variables et aux relations des propriétés qui sortent du cadre mathématique ci-dessus, mais qu'il est indispensable de gérer dans le processus de modélisation/simulation.

- Ainsi, représentant une grandeur physique, chaque variable est associée à une *unité* physique, et à une *échelle* ; on peut aussi lui attacher une précision ou une *incertitude* – voire un modèle de bruit – s'il s'agit d'une entrée. Les équations du modèle doivent permettre de vérifier la cohérence des propriétés connues des variables, et de propager ces propriétés le cas échéant.
- Mais s'agissant d'équations modélisant des relations entres des grandeurs physiques, elles possèdent aussi des propriétés théoriques qui leur sont propres, comme la précision de leur approximation, leur domaine de validité ou leur sensibilité aux paramètres d'entrée. Là encore, il faut être capable d'assurer la cohérence de ces propriétés et leur propagation à travers le modèle.

Au cours du développement des prototypes Lacapio et Ypama, on s'est bien entendu intéressé à gérer ces propriétés des variables et des équations des modèles, à les analyser et à vérifier leur cohérence. Ces points ont été plus particulièrement abordés dans le rapport technique suivant
[58] *ypama : un assistant à la modélisation, analyse dimensionnelle et cohérence des unités.* Rapport d'étude, INRIA/Airbus France, April 2004.
On y reviendra dans la deuxième partie de ce mémoire.

De plus, l'ingénieur a l'habitude de considérer le temps comme une variable à part, et de traiter séparément les problèmes qui en sont indépendants qu'il qualifie de *statiques* et les problèmes qui la font intervenir qu'il qualifie de *dynamiques*. Souvent une discrétisation de la variable temporelle est utilisée pour transformer un modèle dynamique en plusieurs instances d'un problème statique que l'on considère itérativement à chaque pas de temps. Cette approche nécessite de distinguer parmi les variables celles qui sont constantes et donc uniquement définies pour toutes les instances de celles qui représentent des grandeurs qui évoluent au cours du temps, et doivent donc être considérées pour chaque instance, tout en conservant l'information que chacune représente la même grandeur. Le plus fréquemment, le comportement d'un système dynamique à un instant donné dépend aussi de son comportement aux instants précédents, ainsi on verra souvent dans un modèle des variables représentant la même grandeur à des pas de temps différents, ce qui nécessitera pour la simulation de gérer un contexte global extérieur aux différentes instances.

Cette propriété de certaines variables de certains modèles nécessite une adaptation pas forcément simple des algorithmes d'orientation des modèles ou de propagation des propriétés à travers les modèles que nous avons évoqués précédemment.
Si le premier prototype d'environnement de modélisation que j'ai développé pour Airbus, Ypama, se limitait aux modèles statiques, le second, Mosela permet l'étude des modèles de systèmes dynamiques. Il est décrit notamment dans le rapport technique suivant
[59] *The Mosela Modeling and Simulation Environment.* Rapport d'étude, INRIA-Airbus France, Janvier 2007.

1.3 De la modélisation à la simulation

On a vu comment le processus de modélisation/simulation se décompose en l'identification des grandeurs et des phénomènes physiques, le choix d'un formalisme pour l'élaboration d'un modèle puis l'analyse de ce modèle que l'on oriente et ordonne pour le rendre calculatoire. On peut ajouter à ce processus une étape préliminaire de génération de modèles à partir d'une description du système à modéliser – qui sera réalisée dans un langage ad-hoc – et des choix de formalismes établis. Il reste ensuite à traduire ce modèle calculatoire en un programme de simulation dans un langage adéquat, en l'implémentant de sorte qu'il soit précis, juste et efficace, pour simuler le comportement du système.

Si l'évaluation des performances, et donc la validation des choix techniques de conception, se focalisent sur les résultats de simulation, encore faut-il que les résultats numériques obtenus traduisent de manière fidèle le comportement futur du système que l'on est en train de concevoir. Or, à toutes les étapes du processus de modélisation sont faits des choix de méthodes, assumées des hypothèses, et parfois introduites des incertitudes, qui conditionnent la qualité de ces résultats. La confrontation des résultats ainsi obtenus avec des expérimentations, ou leur comparaison avec des résultats similaires peut conduire à modifier certains des choix de méthodes ou certaines des hypothèses de modélisation et à itérer ce processus de modélisation en trois étapes produisant successivement le modèle formel, le modèle calculatoire, et le code de simulation (voir figure 1.1). La première étape relève du domaine de compétences de l'ingénieur métier qui fait le choix du formalisme et de la théorie à mettre en oeuvre, la deuxième relève des compétences du mathématicien ou du numéricien qui mets en place les schémas de calculs (d'intégration, de résolution, etc...), la troisième fait appel aux compétences de l'informaticien qui optimise la structure des données et des calculs pour obtenir un code efficace.

Avoir au sein d'un même environnement de travail la possibilité de représenter ces deux modèles et ce code, et avoir des outils permettant d'automatiser, au moins partiellement les passages entre ces représentations, conditionne une bonne collaboration entre l'ingénierie métier nécessaire à l'élaboration du modèle physique, l'ingénierie mathématique permettant de l'expliciter, et l'ingénierie informatique en assurant une implémentation rigoureuse et

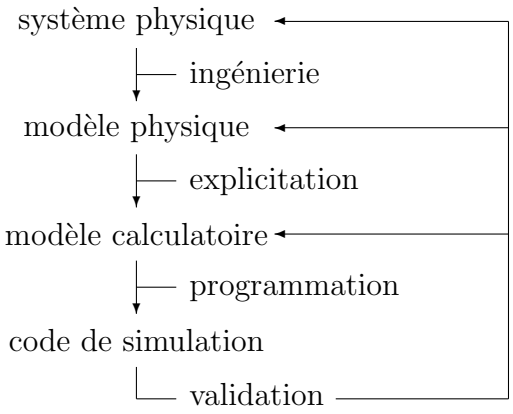

FIGURE 1.1 – Processus de modélisation/simulation

efficace. Cela permet notamment d'effectuer les itérations du processus beaucoup plus simplement, et donc d'être plus exigeant, à moindre coût, envers la qualité des résultats obtenus.

De fait, dans le monde industriel d'aujourd'hui, les outils n'existent pas qui permettraient, dans un environnement commun, que des experts de chaque domaine de compétence – les compétences métier, l'analyse numérique, l'informatique – interviennent successivement pour produire un code de simulation efficace d'un système physique. Par contre, de nombreux logiciels de simulation sont disponibles, dont la plupart s'appuient sur des formats de description et de représentation de modèles qui leur sont propres, et produisent un code de simulation plus ou moins efficace en fonction d'une description plus ou moins équationnelle du système étudié.

> Un état de l'art des outils intégrés de modélisation et de simulation est présenté dans [45] *Modélisation et simulation avec Mathematica.* Conférence Mathematica Paris 2004.

L'obtention d'un code de simulation numérique précis et efficace du comportement d'un système complexe à partir de la description du modèle de ce système dans un langage de haut niveau proche du métier de sa physique, peut s'assimiler à un processus de *compilation* de ce langage source symbolique vers un langage cible comme C ou Fortran et les techniques et les méthodes utiles à sa mise en oeuvre sont voisines de celles utilisées dans ce domaine de l'informatique.

On peut pousser l'analogie en rapprochant cette notion d'environnement de modélisation et de simulation d'un environnement de développement informatique – à la Eclipse, par exemple – qui intègre à la fois des outils d'édition, de vérification syntaxique et sémantique, de compilation, d'exécution du code cible, de visualisation des résultats, de mise au point, etc.

Les fonctionnalités et les spécifications fonctionnelles d'un tel environnement de modélisation/simulation seront longuement développées dans la seconde partie de ce mémoire. Elles ont aussi fait l'objet du rapport technique
[59] *The Mosela Modeling and Simulation Environment.* Rapport d'étude, INRIA-Airbus France, Janvier 2007.

1.4 Quelques contributions

Etudier les méthodes pour la modélisation et la simulation à partir de descriptions symboliques des équations qui gouvernent le comportement des systèmes a été un thème central de mon activité scientifique des 25 dernières années. Les paragraphes ci-dessous présentent les résultats applicatifs obtenus en un exposé thématique et chronologique.

1.4.1 En mécanique

De 1987 à 1992, j'ai travaillé, en collaboration avec l'aerospatiale Cannes, sur l'étude du comportement dynamique de systèmes de corps rigides poly-articulés. L'utilisation de méthodes et d'outils de calcul formel, très peu développée à l'époque, a permis d'attaquer deux problématiques intéressantes :
– la génération automatique du modèle équationnel
– la prise en compte de bouclages structurels

génération du modèle équationnel Avec un champ applicatif réduit à l'étude de mécanismes rigides embarqués sur des satellites et destinés au déploiement et au positionnement d'outils ou de capteurs, il était envisageable de dresser un catalogue exhaustif de composants et de liaisons génériques de ces mécanismes. Il s'agissait donc de voir comment les équations de la dynamique du système pouvaient se déduire des équations de chaque composant et de la structure topologique du système en termes de liaisons, ainsi que de la nature de ces liaisons. Nous avons montré comment l'ajout d'un composant à un système existant, se traduisait, dans l'approche Lagrangienne choisie, par l'adjonction de paramètres de configuration et le recalcul systématique d'une matrice de contrainte ne dépendant que de la nature de la liaison considérée. Ainsi, un processus itératif d'ajout de composant permettait de construire complètement automatiquement les équations algébro-différentielles de Lagrange du système, à partir de sa description structurelle.

prise en compte de bouclages structurels L'exhibition d'un paramétrage minimal de la variété de configurations du mécanisme fournit un noyau de la matrice de contrainte qui représente la partie algébrique des équations de Lagrange, et permet ainsi l'élimination formelle des multiplicateurs de Lagrange dans ces équations. Nous avons montré que la

détermination automatique d'un tel paramétrage n'est possible que lorsque la structure topologique des liaisons entre les composants du mécanisme est arborescente. Dans le cas de bouclages structurels, une étude algébrique de la matrice de contrainte, et notamment du module de syzygies associé, a permis, dans certains cas la détermination du nombre de degrés de liberté et l'élimination des multiplicateurs.

Ces travaux sont décrits dans ma thèse de doctorat, et dans les articles et rapports suivants :
[1] *Outils formels pour la modélisation en mécanique.* PhD thesis, Université de Nice Sophia-Antipolis, France, November 1992.
[48] avec C. Garnier et P. Rideau. *Modélisation dynamique littérale.* Rapport technique 1255 CA/TC, aerospatiale, March 1987.
[9] avec C. Garnier et P. Rideau. *Modélisation dynamique littérale.* Journal of Computer Methods in Applied Mechanics and Engineering, 75 :215–225, 1989.
[49] avec P. Capolsini, L. Pottier, et P. Rideau. *Outils d'aide à la modélisation formelle en mécanique et en automatique.* Rapport Technique CA/TSV 173, aerospatiale/INRIA, December 1991.
[44] *Calcul formel pour la modélisation en mécanique.* In *CALSYF, 9.* GRECO de Calcul Formel, 1991.

1.4.2 En électronique

Mes travaux sur la génération de modèle de simulation de transistors MOS on été réalisé en collaboration avec Siemens, Munich en 1995.

La taille du modèle en jeu et le nombre de paramètres intervenant dans les calculs conduisant à des expressions littérales trop volumineuses et à des pertes de performances ou à des blocages logiciels, nous nous sommes penchés sur la recherche de sous-expressions dans les expressions et sur la représentation des expressions par des programmes de calcul. Nous nous sommes intéressés à diverses manipulations sur les expressions ainsi représentées et notamment à leur différentiation en combinant des techniques issues de la *différentiation automatique* et des techniques de différentiation symbolique.

Ces travaux ont été présentés dans
[15] avec W. Klein. *Automatic generation of a mos transistor simulation model with maple v.* Nuclear Instruments and Methods in Physics Research A, 389 :125–127, 1997.

1.4.3 En optique

J'ai étudié, en 1995 et en 1996, la modélisation des performances optique d'un interféromètre destiné à être embarqué sur un satellite d'observation, dans le cadre d'une autre

collaboration avec l'aerospatiale Cannes. Ces travaux ont débouché sur la réalisation d'une thèse[Phil97] et d'un prototype CIRCE.

Dans cette étude, c'est l'automatisation du processus de génération du modèle calculatoire à partir du modèle physique qui a été principalement investiguée et notamment l'automatisation des étapes d'orientation et d'ordonnancement des équations.

En effet, en pleine phase de conception et de dimensionnement de cet instrument optique, et étant donné le grand nombre de paramètres en jeu, la multiplicité et la diversité des modèles reliant ces paramètres entre eux (mécaniques, optiques, électronique, électromagnétique, thermique, etc.), il était essentiel de pouvoir répéter facilement et rapidement le scénario suivant :

1. identifier comme entrées et comme sorties de la simulation certains paramètres,
2. vérifier la possibilité de calculer les sorties en fonction des entrées (ou revenir à l'étape précédente)
3. générer et mettre en oeuvre le code de simulation correspondant

Ces travaux sont décrits dans les articles et les rapports suivants :
[13] avec S. Philoreau. *Design and performances analysis of complex optical devices using symbolic computation.* In Scientists Inc., editor, *Proceedings of the 2nd ASCM*, pages 55–61, Tokyo, August 1996. H. Kobayashi Ed.
[26] avec S. Dalmas. *Design and performance analysis of optical instruments with computer algebra.* In University of New Mexico, editor, *Electronic proceedings of the first IMACS conference on Applications of Computer Algebra*, 1995.
[27] avec S. Philoreau, D. Miras, et D. Simeoni. *Circe : A new approach to performance management of optical instruments.* In SPIE, editor, *Proceedings of the SPIE 96 Annual Meeting, volume 2817.* Infrared Spaceborne Remote Sensing Conference, August 1996.
[28] *From modeling to simulation with symbolic computation : An application to design and performance analysis of complex optical devices.* In *Proceedings of the Second Workshop on Computer Algebra in Scientific Computing.* Munich, June 1999. Springer, Telos.
[51] *Un logiciel d'aide à la conception et à l'évaluation des performances d'instruments optiques.* Rapport Technique, aerospatiale/INRIA, février 1995.
[52] *Aide à la conception et à l'évaluation des performances d'instruments optiques.* Rapport Technique, aerospatiale/INRIA, mars 1996.

1.4.4 En aéronautique

Mes travaux de modélisation aérodynamique et mécanique en aéronautique sont réalisés en proche collaboration avec Airbus depuis 2002.

environnement de modélisation Lorsque démarre cette collaboration, la chaîne de développement d'une nouvelle génération d'avion est particulièrement complexe, elle fait intervenir de nombreux acteurs au sein du bureau d'étude qui produisent ou raffinent les modèles aérodynamiques et dynamiques de l'avion sous forme de documents imprimés et qui les transmettent à leur collègues informaticiens en charge du développement du simulateur de vol dédié. L'ambition de l'étude est d'automatiser ce processus de génération des simulateurs en dotant Airbus d'un environnement d'édition de modèles qui permette à la fois leur validation numérique expérimentale et leur traduction en un code numérique d'évaluation efficace intégrable dans les simulateurs de vol existant.

La plupart des fonctionnalités nécessaires existe à l'état embryonnaire dans les outils logiciels de calcul formel, mais nécessiteront d'être développées et intégrées. Le prototype Ypama développé sera repris et industrialisé par Airbus en 2006 pour entrer (sous un autre nom) dans sa chaîne de production.

Au cours de cette étude, nous nous sommes beaucoup intéressés à la propagation de certaines propriétés des variables au travers des modèles, notamment les unités physiques, les échelles, et les domaines de validité, ce qui a conduit à utiliser et à développer si nécessaire des arithmétiques exotiques sur ces propriétés.

Peu de publications concernent ces travaux qui ne devaient pas être divulgués avec trop de détail pour des raisons de confidentialité industrielle. Toutefois ils sont décrits dans les rapports suivants :
[33] avec L. Farenc. *Modeling flight dynamics for real-time simulator applications.* In *Applied Mathematica, Proc. of the 8th Int. Mathematica Symposium*, Juin 2006.
[56] *ypama : un assistant à la modélisation.* Rapport d'étude, INRIA/Airbus France, Janvier 2002.
[57] *ypama : un assistant à la modélisation, comment implémenter la désorientation des modèles.* Rapport d'étude, INRIA/Airbus France, December 2003.
[58] *ypama : un assistant à la modélisation, analyse dimensionnelle et cohérence des unités.* Rapport d'étude, INRIA/Airbus France, April 2004.

génération de code C Au sein d'un autre département du bureau d'étude, une autre collaboration est initiée cette même année qui se place thématiquement dans la continuité de la première mais cible plus particulièrement la génération complètement automatique de code de simulation en langage C à partir de descriptions équationnelles des modèle et l'extension du champ d'application de l'environnement aux modèles dynamiques.

Le développement d'un deuxième prototype intégrant environnement d'édition, génération de code d'évaluation numérique pour la mise au point, génération de code de simulation numérique en langage C et production automatique de documentation métier est sur le point de se terminer.

Là encore et pour les mêmes raisons, peu de publications concernent ces travaux décrits essentiellement dans les articles et rapports suivants :
[33] avec L. Farenc. *Modeling flight dynamics for real-time simulator applications.* In *Applied Mathematica, Proc. of the 8th Int. Mathematica Symposium*, Juin 2006.
[59] *The Mosela Modeling and Simulation Environment.* Rapport d'étude, INRIA-Airbus France, Janvier 2007.
[60] *The Mosela Modeling and Simulation Environment : Code Generation.* Rapport d'étude, INRIA-Airbus France, Avril 2008.
[61] *The Mosela Modeling and Simulation Environment : Model Verification.* Rapport d'etude, INRIA-Airbus France, Avril 2009.
[62] *The Mosela Modeling and Simulation Environment : Efficient Compilation.* Rapport d'etude, INRIA-Airbus France, December 2010.
[63] The Mosela Modeling and Simulation Environment : User's Guide. Rapport d'etude, INRIA-Airbus France, November 2011.

Chapitre 2

Efficacité, précision et exactitude

L'efficacité, la précision et l'exactitude sont les trois qualités que l'on demande principalement à un calcul. Et le fait d'utiliser une arithmétique essentiellement exacte, manipulant des entiers et des rationnels conjointement avec des variables, combiné à une évaluation numérique en précision arbitraire, semble autoriser les outils de calculs formels à garantir ces qualités. Mais l'on va voir que l'efficacité n'est pas toujours au rendez-vous, et qu'il faut prendre quelques précautions dans l'interprétation des résultats obtenus si l'on veut conserver la précision et l'exactitude. On verra aussi que les méthodes formelles n'ont pas l'apanage de la précision et de l'exactitude et que d'autre méthodes s'appuyant sur les intervalles garantissent l'exactitude et la qualité numérique de leurs résultats.

2.1 Calcul symbolique versus calcul numérique

Le monde académique a longtemps opposé, au moins en tant que domaines de recherche, calcul formel à calcul numérique et de fait, il a fallu attendre le début des années 2000 pour que des outils informatiques de calculs combinent des fonctionnalités symboliques et des fonctionnalités numériques intéressantes et performantes.

Pourtant ces deux types de calculs sont intrinsèquement imbriqués et indissociables, tant ils font l'objet de liens variés. Sur le plan de l'utilisation, tout d'abord, quel est le calcul numérique qui ne se conçoit pas au départ par des formules et des équations, et quel est le calcul formel que l'on ne cherchera pas à visualiser par une évaluation numérique voire un tracé ? Sur le plan théorique aussi, la nature des objets numériques représentés par des variables conditionnent les traitements que l'on peut leur appliquer – par exemple $\sqrt{x^2}$ ne se simplifiera en x que si la valeur de x est positive, et $x - x$ ne devra pas s'évaluer en 0 si x représente un intervalle – tandis que l'étude symbolique d'un objet permettra de mieux connaître la précision ou la justesse de son évaluation numérique. Enfin en ce qui concerne les méthodes, c'est souvent une discussion ou une classification symbo-

lique qui détermine la nature de la méthode numérique à utiliser, et c'est souvent un pré-conditionnement symbolique qui garantit l'efficacité ou la convergence d'algorithme numériques itératifs. Symétriquement, ce sont parfois de simples et rapides évaluations numériques qui permettent de franchir des étapes décisives dans un algorithme symbolique – ainsi, une évaluation aléatoire des variables permettra de déceler l'inégalité de deux expressions symboliques qui ne serait apparue évidentes qu'après de longues et coûteuses transformations.

Mais le premier de ces liens est historique : c'est bel et bien le besoin de pouvoir faire exécuter à une machine des calculs numériques toujours plus précis, plus complexes et donc gourmands en temps et en mémoire qui a conduit au développement des premiers outils de calcul symbolique, le plus souvent sous l'impulsion des physiciens. Historiquement, c'est en effet dans l'entourage des experts de mécanique céleste, de relativité générale ou d'électrodynamique quantique qu'ont vu le jour les premiers systèmes de calculs formels, pour manipuler les interminables développement en séries des uns, les dizaines de milliers de termes des équations des autres, et intégrer les énormes diagrammes des derniers.

En permettant de manipuler des expressions et des variables, l'environnement de calcul symbolique attire l'utilisateur en lui donnant l'espoir de résoudre des problèmes d'une autre dimension sur le plan de la difficulté, mais sans le lien avec le calcul numérique, il ne lui permettrait pas d'exploiter ni d'appliquer ces résultats.

J'ai présenté et illustré ces considérations notamment historiques lors d'une invitation à une conférence :
[24] *Le calcul formel : un outil pour l'ingénieur.* In Universitat Politecnica de Catalunya, editor, *TEMU-95*, February 1995.

2.1.1 Apports et limitations du calcul symbolique

Pour la plupart des méthodes numériques, intrinsèquement approximantes, on peut généralement exhiber des exemples qui les mettent en défaut, soit au niveau de la convergence, soit au niveau de la justesse du résultat retourné.

La figure 2.1 ci-dessous montre comment l'application répétée d'un processus de résolution de Newton pour le calcul d'une trajectoire le long d'une courbe solution peut conduire à des résultats erronés, ce processus n'apportant ni garantie ni même indication sur la solution vers laquelle il converge.

Sur cette figure, les courbes grises représentent les deux composantes de la courbe des solutions, qui correspondent respectivement à deux trajectoires possibles. La courbe noire est la trajectoire calculée par l'itération, en des points d'abscisses différentes, de prochessus de résolution de Newton. On voit que cette trajectoire "saute" d'une composante solution à l'autre lorsque les deux composantes sont proches. De plus ces courbes grises peuvent

être choisies arbitrairement proches et donc l'augmentation de la précision numérique ne permettra pas d'éviter ce phénomène.

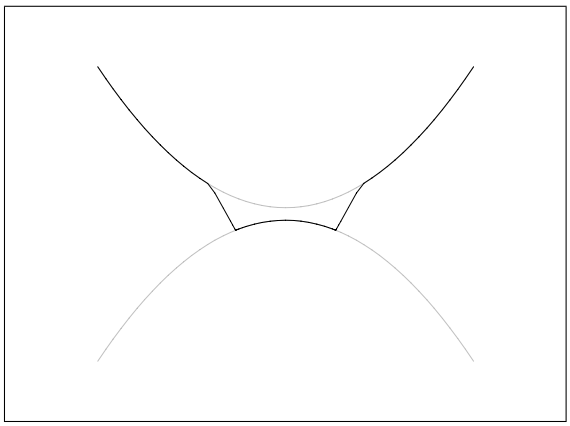

FIGURE 2.1 – saut entre deux composantes solutions

Sur cet exemple, un calcul symbolique fournirait une description explicite de chacune des composantes ce qui résoudrait le problème. Un suivi par évaluations numériques successives d'une des composantes permettrait alors de visualiser cette trajectoire, juste, et à coup sûr.

Toutefois, un inconvénient connu de nombreux algorithmes de calculs symboliques est la taille des expressions intermédiaires qu'ils engendrent, ce qui rend les calcul parfois impraticables, car excédant les capacités mémoires des machines, ou parfois produit des résultats trop gros (en nombre de termes) pour être évalués de façon fiable.

Ces phénomènes d'impraticabilité des calculs et d'erreurs d'évaluation en présence de grosses expressions est étudié en détail dans l'article ci-dessous. En considérant la cinématique de mécanismes de suspension automobile de plus en plus complexes, on montre que si les méthodes purement numériques fournissent, sur ces exemples, des résultats inexacts, les méthodes symboliques obtiennent d'abord facilement puis avec peine les résultats et enfin qu'elles échouent sur les mécanismes les plus élaborés.
[4] avec J-P. Merlet, et D. Daney. *Exact kinematics analysis of car's suspension mechanisms using symbolic computation and interval analysis.* Mechanism and Machine Theory. 40, 395-413. 2005.

Le problème de performance pur de l'évaluation numérique de grosses expressions peut souvent être contourné en produisant de manière automatique un code de calcul numérique dédié – par exemple en C ou en FORTRAN – qui sera un ordre de grandeur plus rapide. Mais cette solution ne résout pas le problème de fiabilité de l'évaluation numérique de grosses expressions qui provient de l'accumulation d'erreurs d'arrondis, surtout en présence de valeurs numériques de différents ordres de grandeurs. L'étude d'applications montre qu'il

est souvent préférable de diviser le calcul, même symbolique, en étapes successives qui seront ensuite évaluées numériquement les unes après les autres : il s'agit en quelque sorte de produire un programme de calcul symbolique, séquence d'expressions, plutôt qu'une seule expression.

Ces problèmes de taille des expressions symboliques et leur résolution par des découpages en programmes de calcul ont été largement rencontrés et abordés dans l'étude sur la simulation des transistors MOS décrites section 1.4.2.
[15] avec W. Klein. *Automatic generation of a mos transistor simulation model with maple v.* Nuclear Instruments and Methods in Physics Research A, 389 :125–127, 1997.

2.1.2 Apports et limitations du calcul numérique

Après avoir opposé calculs numériques et calculs symboliques et avoir énuméré les apports du calcul symbolique, il n'est pas nécessaire de revenir sur les limitations du calcul numérique. Citons juste un petit exemple qui montre combien il est facile de rencontrer un problème de précision et de convergence :

> Soit f la fonction numérique réelle définie par $f(x) = 11x - 2$. On a, à n'en pas douter, $f(y) = y$ pour $y = 0.2$. Si l'on note
>
> $$f^{(n)}(x) = \underbrace{f(f(f(\ldots f}_{n \text{ fois}}(x)\ldots)))$$
>
> on devrait avoir $\forall n, f^{(n)}(0.2) = f^{(n-1)}(0.2) = \ldots = 0.2$.
> Mais, en effectuant le calcul numériquement en double précision sur une machine possédant une arithmétique codée en 64 bits, on obtient

n	$f^{(n)}(0.2)$
1	0.2
...	...
12	0.200005
13	0.200051
14	0.200557
15	0.206132
16	0.267457
17	0.942028
18	8.36231
19	89.9854
20	987.84
21	10864.2

Toutefois, toujours dans le but de calculer efficacement, précisément et juste, on va voir comment le calcul numérique peut nous permettre de dépasser les limitations du calcul symbolique, notamment au travers des algorithmes symboliques–numériques, et à travers les méthodes de l'analyse par intervalles.

2.2 Algorithmes symboliques–numériques

L'apparition et le développement des algorithmes symboliques-numériques se sont faits conjointement et de manière complémentaire au sein des communautés académiques de calcul formel et de calcul numérique.

C'est depuis les années 90 que la communauté académique de calcul formel, et notamment la composante algébriste de cette communauté s'intéresse à la possibilité d'employer du calcul numérique à l'intérieur d'algorithmes de résolution symboliques pour contourner les étapes trop coûteuses en temps de calcul ou trop gourmande en mémoire en raison de la taille et du nombre des expressions intermédiaires générées. Depuis lors de nombreuses équipes se sont penchées sur ces méthodes, pour évaluer la possibilité d'utiliser par endroit une arithmétique en précision finie et des approximations de rationnels par des nombres flottants à l'intérieur d'algorithmes de calcul formel.

Un exemple caractéristique est le calcul de zéros de polynômes pour lequel le recours à des calculs de valeurs propres permet d'éviter des développements de résultants extrêmement couteux [Mou98].

En continuation d'une étude commencée lors de mon séjour post-doctoral à ETL, j'ai mis en oeuvre de tels algorithmes pour le calcul de la silhouette d'un ensemble semi-algébrique. Ce calcul intervient dans ce cas notamment dans la planification de mouvement de robots en contacts.
[17] avec H. Hirukawa et B. Mourrain. *A symbolic-numeric silhouette algorithm.* In Proceedings of 2000 IEEE International conference on RObotics Systems.

Pendant ce temps, la communauté académique d'analyse numérique s'intéresse à l'emploi d'outils et de méthodes de calcul formel notamment pour appliquer un pré-conditionnement à certains problèmes mal conditionnés et les rendre traitables. Dans la résolutions de systèmes d'équations polynomiales, ou matricielles à coefficients paramétriques, Il s'agit par exemple d'essayer de factoriser ou de simplifier analytiquement les équations à résoudre, par des combinaisons, par des substitutions, ou par des changements de variables.

Nous avons ainsi utilisé le calcul formel pour analyser et simplifier des ensembles d'équations de distances qui interviennent dans la modélisation des robots parallèles, notamment lors de calculs de positionnement de robots en présence d'incertitudes.
[21] avec C. Grandon, D. Daney, C. Tavolieri, E. Ottaviano, et M. Ceccarelli. *Handling Uncertainties with Symbolic/Numerical Solvers for a Class of Parallel Robots*. In WC - International Federation for the Theory of Machines and Mechanisms (IFToMM), June 2007.

2.3 Analyse par intervalles

Intrinsèquement juste malgré son caractère numérique - à condition de bien tenir compte des arrondis effectués par les calculateurs en nombres flottants - l'arithmétique par intervalles combine à la fois l'efficacité du calcul numérique et l'exactitude du calcul symbolique, mais au prix d'une perte de précision qui se propage en empirant au long des calculs du fait, entre autres, de l'accumulation de ces erreurs d'arrondis.

Une autre raison de la perte de précision dans les calculs est que dans cette arithmétique, ni l'addition, ni la multiplication n'ont les propriétés classiques d'éléments neutres et symétriques et que donc $x - x$ n'est pas nul si x est un intervalle :

$$[0, 1] - [0, 1] = [-1, 1]$$

Enfin un effet d'enveloppement dont deux exemples sont représentés dans la figure 2.2 se manifeste sur les calculs multidimensionnels. Cet effet concourt à surestimer les résultats de calculs et encore une fois à perdre en précision. Sur la figure on voit teintées en rose les zones calculées par des méthodes d'analyse par intervalles pour localiser la courbe dans le dessin de droite et l'aire d'intersection des deux couronnes dans le dessin de gauche. Sont aussi représentés par un contour en gras les ensembles que l'on pourrait espérer obtenir en supprimant l'effet d'enveloppement par un changement d'axes ad-hoc.

Une communauté scientifique dont Moore [Moo65] fut un pionnier en 1960, a développé des méthodes d'évaluation d'expressions, et adapté des algorithmes numériques de résolution et d'optimisation spécifiquement pour compenser ces pertes de précision. Par la suite, la combinaison de ces méthodes avec des techniques de propagation et de filtrage issues de la programmation par contraintes a produit des algorithmes encore plus efficaces et robustes aux incertitudes.

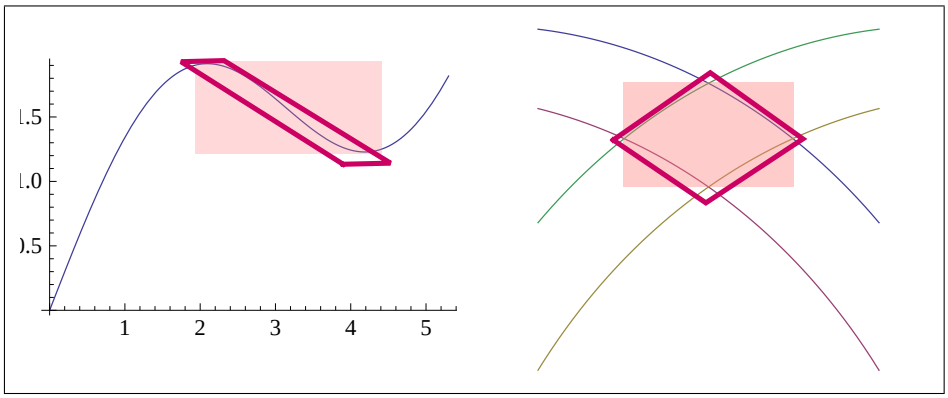

FIGURE 2.2 – effet d'enveloppement

Ma contribution à ces méthodes dites d'*analyse par intervalles* dont le développement a connu un essor au début des années 2000 porte à la fois
- sur leur diffusion, en développant l'interface entre la bibliothèque ALIAS d'analyse par intervalles et le logiciel *Mathematica*,
  [30] avec D. Daney. *Solving with interval analysis.* In *Wolfram Technology Conference*, Illinois, USA, 2005.
- sur leur utilisation, dans le cadre de la modélisation cinématique de robots parallèles, pour laquelle elles se sont avérées très efficaces,
  [32] avec C. Grandón et D. Daney. *Combining CP and interval methods for solving the direct kinematic of a parallel robot under uncertainties.* In Workshop of the Int. Conf. on Principles and Practice of Constraint Programming (IntCP 06), September 2006.
- sur leur développement, notamment pour leur extension à l'intégration numérique d'équations différentielles,
  [39] avec N. Ramdani. *Guaranteed Numerical Integration of Nonlinear Parametric ODEs.* In Proceedings of the 9th International Mathematica Symposium, Maastricht, The Netherlands, 2008.

Calcul symbolique et analyse par intervalles Un corollaire de l'absence d'élément neutre pour l'addition des intervalles est la sensibilité de l'évaluation d'une expression à sa forme analytique.

Ainsi une évaluation du polynôme $P(x) = x^2 + x - 6$ sur l'intervalle $[-1, 1]$ donne $[-7, -4]$. Or $P(x)$ peut s'écrire de manière symboliquement équivalente :
- en factorisant $(x-2)(x+3)$ qui s'évalue en $[-12, -2]$,
- sous forme de Horner $x(x+1) - 6$ qui s'évalue en $[-8, -4]$,
- ou même $(x + \frac{1}{2})^2 - \frac{25}{4}$ qui s'évalue en $[-6.25, -4]$

Cet exemple montre bien l'influence d'un pré-conditionnement symbolique sur la qualité

numérique de l'évaluation.

Le pré-conditionnement des expressions pour l'évaluation n'est qu'un exemple d'utilisation du calcul symbolique en analyse par intervalles. Une autre utilisation est dans l'implémentation d'un algorithme de filtrage issu de la programmation par contrainte qui nécessite de multiples résolutions d'une équation par rapport à toutes ses variables. On peut aussi l'utiliser pour améliorer des algorithmes numériques en forçant une pré-exécution symbolique.

Divers pistes ont été explorées dans la communication suivante :
[35] avec D. Daney. *Computing with Intervals : the Ultimate Symbolic Computation ?* In Wolfram Technology Conference, Champaign, Illinois, USA, 2007.

Conclusion

Les sections précédentes ont établi que pour calculer efficacement, précisément et juste, ni le calcul symbolique, ni le calcul numérique ne suffit dans la plupart des cas complexes. Par contre la combinaison de fonctionnalités numériques et symboliques permet de développer des méthodes variées où l'évaluation numérique pallie au manque d'efficacité des calculs symboliques (en temps et en espace), et où l'analyse et le pré-conditionnement symbolique protège des divergences et perte de précisions. Enfin l'analyse par intervalles qui utilise à la fois des méthodes symboliques et numériques pour compenser l'accumulation et la propagation des imprécisions dans les calculs possède toutes les qualités pour effectuer des calculs de simulation.

La prochaine section présente mes travaux dans le cadre des algorithmes symboliques-numériques qui proviennent d'exemple d'application en robotique.

2.4 Quelques contributions : applications en robotique

Parmi les domaines où la modélisation et la simulation sont d'une importance primordiale, la robotique a une place de choix. C'est une science essentiellement tournée vers la conception de systèmes complexes, innovants et originaux, repoussant sans cesse les limites de l'imagination, de la créativité et de la technologie. Ces systèmes sont amenés, le plus souvent, à appréhender leur environnement, à s'y adapter ou à interagir avec lui. Ainsi les modèles sont omniprésents soit pour représenter les systèmes eux-mêmes, soit pour représenter leur environnement, ou encore leur interaction.

Planification de mouvement En séjour post-doctoral au laboratoire de robotique d'ElectroTechnical Lab., j'ai étudié la planification de mouvement d'objets en contacts. Le

but ultime était de calculer une séquence de mouvements d'un effecteur qui aurait permis d'amener un objet maintenu en contact avec cet effecteur sous la seule action de la gravité d'une position initiale mesurée jusqu'à une position terminale prédéterminée. Dans la lignée de travaux de Brooks [Bro83], je me suis intéressé à la modélisation géométrique des différents objets, et de leur espace de configurations cinématiques, utilisant une représentation analytique en termes d'ensembles semi-algébriques puis à la caractérisation de ces ensembles semi algébriques et de leur frontière dans l'optique d'une planification de mouvement en contact,

Les différentes avancées réalisées dans le cadre de cette étude ont été publiées en collaboration avec mon hôte à ETL, le Dr. H. Hirukawa.
[10] avec H. Hirukawa, et T. Matsui. *A motion planning algorithm for convex polyhedra in contact under translation and rotation.* In proceedings of the 1994 IEEE International Conference on Robotics and Automation, 3020–3028, San Diego, California, May 1994.
[11] avec H. Hirukawa et H. Tsukune. *A motion planning algorithm of polyhedra in contact for mechanical assembly.* In proceedings of the 20th International Conference on Industrial Electronics Control and Instrumentation, pages 924–929, Bologna, Italia, September 1994.
[12] avec H. Hirukawa. *A lazy algorithm for planning motions in contact.* In proceedings of the 1994 IEEE/RSJ/GI International Conference on Intelligent Robots and Systems, 2152–2159, Munich, Germany, September 1994.
[16] avec H. Hirukawa. *Motion planning of objects in contact by the silhouette algorithm.* In proceedings of the 2000 IEEE International Conference on Robotics and Automation, 3020–3028, San Francisco, California, May 2000.

Calibration de robots parallèles En 2002, j'ai été étroitement mêlé à la création de l'équipe-projet COPRIN de l'INRIA. Sous l'impulsion de Jean-Pierre Merlet [Mer93], cette équipe, dont l'acronyme signifie "Contraintes, OPtimisation, Résolution par INtervalles" a privilégié l'étude des robots parallèles et des robots à câbles comme domaine applicatif et comme source de problème pour expérimenter ses méthodes.

Parmi les multiples problèmes abordés par l'équipe, allant de la modélisation directe et inverse cinématique et dynamique aux calculs d'espaces de travail en passant par la conception optimale, je me suis essentiellement intéressé aux problèmes de calibration.

Si la calibration s'appuie sur les modèles formels classiques des robots étudiés, elle les considère habituellement avec une orientation différente, les paramètres à déterminer étant les paramètres physiques du robot étudié, à partir de positions mesurées de la base et de l'effecteur. De plus la calibration fournit en général des systèmes d'équations surcontraints, le nombre de mesures réalisées étant largement supérieur au nombre de paramètres à déterminer. Ces problèmes conviennent bien à certaines des méthodes de calculs issues de l'analyse par intervalles qui voient les équations à résoudre comme autant de contraintes à satisfaire au mieux en fonction des incertitudes. Plusieurs expériences ont donné lieu à divers publications.

[5] avec D. Daney et B. Madeline. *Choosing Measurement Poses for Robot Calibration with Local Convergence Method and Tabu Search.* International Journal of Robotic Research, 24(6) :501–518, 2005.

[6] avec D. Daney, N. Andreff, et G. Chabert. *Interval method for calibration of parallel robots : a vision-based experimentation.* Mechanism and Machine Theory, 41(8) :929–944, August 2006.

[19] avec D. Daney, et A. Neumaier. *Interval methods for certification of the kinematic calibration of parallel robots.* In Proc. IEEE International Conference on Robotics and Automation (ICRA), 147–152, New Orleans, USA, April 28-30 2004.

[20] avec D. Daney et N. Andreff. *Interval method for calibration of parallel robots : a vision-based experimentation.* In Computational Kinematics, Cassino, 2005.

Chapitre 3

Quelques contributions logicielles

Utilisateur quotidien des logiciels de calcul formel Macsyma, puis Maple et Mathematica, j'ai pu apprécié combien en trois décades ces logiciels se sont développés et enrichis.

Ils étaient initialement concentrés sur ce qui faisait leur spécificité à savoir des capacités de manipulation symbolique bien entendu, mais aussi une arithmétique entière, une arithmétique rationnelle exacte et une arithmétique flottante en précision infinie, dont les capacités n'avaient de limites que celles imposées par la puissance de la machine. Et si, en évoluant, ils ont intégré, sous forme d'implémentation, les meilleurs algorithmes de calcul formel issus de la recherche - concernant notamment la résolution d'équations polynomiales, la factorisation et la simplification d'expressions, l'intégration, les séries, le calcul différentiel, etc. - leur évolution la plus notable est de s'être ouverts vers le monde de l'ingénieur et d'être devenus des environnements de calcul intégrés combinant à la fois des fonctionnalités d'import et d'export de données, des routines efficaces d'intégration, de résolution et d'optimisation numérique, une puissance de calcul matriciel honorable, d'excellentes fonctionnalités d'analyse statistique et de visualisation de données, et d'avoir élargi la nature des objets qu'ils peuvent manipuler aux graphiques, images, sons et vidéos.

J'ai accompagné cette évolution en développant, au sein de logiciels comme Maple ou Mathematica plusieurs bibliothèques pour répondre généralement a un besoin ponctuel que je ressentais et ainsi ajouter une fonctionnalité manquante. Il est arrivé que ces fonctionnalités soient ensuite, ou en même temps, développées par les concepteurs de ces systèmes et que ces bibliothèques deviennent obsolètes. D'autres fournissent des fonctionnalités qui n'existent toujours pas ou ont été intégrées dans ces systèmes.

La suite de ce chapitre présente mes contributions logicielles les plus significatives dans quatre domaines essentiels pour les logiciels de calcul formel : la représentation des expressions, la communication inter-systèmes, la génération de code de bas niveau, et les algorithmes de calcul.

3.1 Représenter les expressions

Dans le cadre d'une étude réalisée en collaboration avec le Centre National d'Etude Spatiale concernant la cinématique de systèmes mécaniques complexes et poly-articulés, j'avais été amené à utiliser des quaternions pour représenter les orientations relatives des repères attachés aux divers composants mécaniques et la modélisation conduisait à des expressions de grandes tailles faisant intervenir des quaternions.

Un quaternion peut être représenté selon le cas par la donnée de quatre coordonnées par rapport à une base canonique obéissant à des règles de calcul exotiques : le corps des quaternions est un corps non commutatif que l'on peut considérer comme engendré par $1, e_1, e_2, e_3$ vérifiant $\frac{1}{2}(e_i e_j + e_j e_i) = \delta_{ij}$. Mais on l'utilise plus souvent sous la forme d'une paire composé d'un scalaire et d'un vecteur à cause de l'isomorphisme entre l'ensemble des rotations de l'espace et celui des quaternions unitaires.

De plus, un autre isomorphisme entre l'ensemble des quaternions et une sous-algèbre des matrices 2×2 à coefficients complexes nous apporte - par transposition de l'identité de Cayley-Hamilton un lot de règles de simplifications qui s'applique sur les symboles lorsque ceux-ci représentent des quaternions. Ainsi, par exemple, si q_1, q_2 et q_3 sont des quaternions et si on note $< q_1, q_2 >= q_1 q_2 - q_2 q_1$, on a $<< q_1, q_2 >^2, q_3 >= 0$.

Il est donc important de pouvoir appliquer ces règles de calcul et de simplifications, lorsque l'on développe puis que l'on simplifie une expression dont les variables représentent des quaternions, et cela avant de passer à une représentation en termes de scalaires et de vecteurs. Ensuite de nouvelles règles s'appliqueront au niveau scalaire et vecteur, et ce n'est qu'en fin de calcul que l'on passera aux coordonnées.

Un système de calcul formel permet généralement de définir de nouveaux opérateurs à partir des règles de calcul qui leur sont associées et d'implémenter des règles de simplification sur les expressions. Cela constitue, avec les outils de passage d'une représentation à l'autre et les outils de conversion entre quaternions et rotations l'essentiels des fonctionnalités des bibliothèques que nous avons développées pour la manipulation des quaternions.

On retrouvera dans le détail toutes ces considérations théoriques sur les quaternions, les règles de calculs et de simplification des expressions, leur utilité à représenter les rotations, et l'intérêt à les utiliser en terme de complexité de calcul dans les publications qui décrivent ces bibliothèques développées pour les deux principaux systèmes de calcul formel.
[50] avec P. Capolsini et S Dalmas. *Quaterman : une bibliothèque maple de calculs et de manipulations sur l'algèbre des quaternions - a library for computing with quaternions in Maple.* Rapport technique 148, INRIA-I3S, jan 1993.
[25] avec S. Dalmas. *Quaternica : A package for manipulating expressions involving quaternions.* In Computational Mechanics publications, editor, *Mathematics with vision : proceedings of the First International Mathematica Symposium*, 291–298, 1995.

Cette communication présente une vision plus orienté sur l'implémentation comparée des bibliothèques dans les deux systèmes.
[14] avec S. Dalmas. *Quaterman vs. Quaternica : a comparative implementation of quaternions in maple and mathematica.* In Scientists Inc., editor, *Proceedings of the 2nd ASCM*, 113–120, Tokyo, August 1996.

3.2 Communiquer

L'initiative *OpenMath* [Abb95] a été initiée dans les années 1990 par les principaux acteurs de la branche "systèmes" de la communauté académique du calcul formel dans le but de mettre en place un standard pour la représentation sémantique des objets mathématiques afin de permettre l'échange de ces objets entre des programmes, avec des bases de données et leur publication notamment sur le Web. Fortement relié au standard MathML développé par le W3C pour la communication des objets mathématiques sur la toile, le format OpenMath se concentre sur la sémantique là où MathML considère une double représentation basée sur la présentation et le contenu. Ainsi OpenMath est un des candidats a fournir un contenu standard aux objets MathML.

Dans le cadre de la mise en place cette initiative, il était important, en marge des activités de normalisations et de standardisations, de développer des prototypes afin de pouvoir créer, échanger, et visualiser des objets à ce format.

Nous avons développé l'interface Mathematica permettant la création, et l'échange de données OpenMath depuis ce programme et mis en place une architecture client-serveur entre deux noyaux de Mathematica communiquant en OpenMath.
[3] avec S. Dalmas et H. Prieto. *Mathematica as an OpenMath application.* SIGSAM Bulletin, 2000.

3.3 Générer du code C

Générer un code de plus bas niveau comme par exemple en langage C pour traduire dans ce langage les algorithmes que l'on a prototypés, ou les calculs que l'on a réalisés dans un langage de calcul formel, est un besoin naturel qui permet d'envisager un gain de performance énorme lors de l'exécution. En effet, si les systèmes de calcul formel sont très permissifs sur le plan de la syntaxe et permettent de manipuler des objets symboliques sans connaître le type des valeurs qu'ils peuvent prendre, si les arithmétiques proposées de manière transparente pour l'utilisateur sont diverses et complexes, ce confort d'utilisation a un coût en terme de performances et le gain de temps qu'il procure au développement a une contrepartie en temps d'exécution du programme une fois terminé.

Générer du code C est un processus en deux étapes qui comporte une phase de traduction des expressions apparaissant dans les instructions élémentaires, et une deuxième phase pendant laquelle il faut analyser la structure du programme formel, lui ajouter des hypothèses, notamment de typage et de représentation des données, et la transposer en un programme C en fabriquant au départ toutes les instructions de déclarations de variables, de gestion et d'allocation de mémoire qui sont inconnues des systèmes de calcul formel.

L'idée originale de posséder un ensemble d'outils à l'intérieur du langage de calcul formel pour effectuer cette deuxième étape du processus de génération de manière algorithmique et calculatrice – et non comme un bricolage de chaines de caractères – est due à C. Gomez [Gom90].

Confronté, dans le cadre d'une collaboration avec Airbus à la difficulté de générer, à partir de Mathematica, du code C destiné à être intégré dans un environnement industriel de simulation de vol et devant satisfaire à des normes de codage et de nommage très spécifiques, j'ai développé une représentation complète du langage C dans le langage de ce système de calcul formel qui permet de représenter toutes les constructions et toute la structure des objets du langage C à la norme ANSI avec leur sémantique. Ainsi le programme C devient dans le processus de génération, le résultat d'une manipulation ou d'un calcul symbolique, et se manipule comme n'importe quel autre objet symbolique jusqu'à une ultime étape d'exportation.

Cette bibliothèque de représentation du langage C en Mathematica s'appelle SymbolicC.
[34] *Industrial C code generation, SymbolicC : a C wrapper.* In Wolfram Technology Conference, Champaign, Illinois, USA, 2006.
[46] *Génération de code C avec Mathematica : le packetage SymbolicC.* In Conférence Mathematica Paris 2007, Juin 2007.

Disponible sur la forge de l'INRIA, SymbolicC a été acquis par Wolfram Research Inc. et a été intégré dans la version 8 de Mathematica après des modifications mineures.

Cette capacité de représenter le code C comme un objet du langage formel permet aujourd'hui a Mathematica de traduire en C à la volée certaines expressions repérées comme coûteuses à évaluer. Les programmes C ainsi générés sont exportés et compilés en des librairies dynamiques. Enfin, et toujours de façon immédiate et transparente pour l'utilisateur, ces librairies sont liées dynamiquement au noyau de calcul. Ce processus améliore sensiblement les performances de certains algorithmes notamment des algorithmes numériques devant faire de nombreuses évaluations de mêmes expressions.

3.4 Calculer

Depuis 1999, sous l'impulsion de Jean-Pierre Merlet, les principales contributions logicielles du projet COPRIN en analyse par intervalles (voir section 2.3) sont implémentées dans la librairie ALIAS [Mer02]. Cette librairie a une composante C++ qui propose de nombreux algorithmes pour résoudre des systèmes d'équations et d'inéquations, résoudre des problèmes d'optimisation globale, d'algèbre linéaire, etc... en travaillant sur tout type d'expression mathématique à base de nombres flottants et d'intervalles.

L'interfaçage de cette librairie avec un logiciel de calcul formel, permet d'une part d'avoir un accès direct depuis ce système aux fonctionnalités de cette librairie, mais il permet surtout d'enrichir cette librairie avec des fonctionnalités de calcul formel, soit pour le pré-conditionnement formel des expressions soit dans l'implémentation d'algorithmes spécifiques (résolutions utilisant le gradient ou le hessien du système d'équations, filtrage 2-B, etc...)

> Pendant d'une interface très complète avec Maple développée par J.P. Merlet, j'ai réalisé une interface d'ALIAS avec Mathematica que je continue à enrichir pour répondre aux besoins des utilisateurs de l'équipe.
> [30] avec D. Daney. *Solving with interval analysis.* In *Wolfram Technology Conference*, Illinois, USA, 2005.

La plupart des algorithmes d'analyse par intervalles peuvent être vus comme des algorithmes de recherche globale sur un domaine de points satisfaisant des contraintes. On peut alors leur donner une forme générique :
- initialiser le domaine de recherche,
- *filtrer* le domaine de recherche par les contraintes (i.e éliminer de ce domaine des sous-ensembles qui ne satisfont pas les contraintes)
- tant que le domaine de recherche n'est pas suffisamment réduit,
  - *partitionner* le domaine en sous-domaines
  - *répéter itérativement* ce processus de recherche sur les sous-domaines

Ce qui fait la richesse et les différences entre ces algorithmes réside dans le choix des méthodes pour chacune de ces trois actions : quelle filtrage utiliser ? comment partitionner le domaine ? comment gérer l'itération et les priorités lors de cette itération ? Pour le concepteur d'algorithme, un quatrième choix concerne la méthode d'évaluation d'expressions utilisée.

On voit alors l'intérêt d'une plateforme de prototypage d'algorithme permettant de rapidement faire ces choix, développer et expérimenter l'algorithme correspondant, en en évaluant les performances non pas en terme de temps de calcul brut, mais de critères tels que le nombre d'évaluations réalisées ou le nombre de sous-domaines considérés.

J'ai réalisé une telle plateforme en Mathematica sous la forme d'un package, UnCertainities, qui lui aussi évolue encore en fonction des besoins.
[36] *Certified Computations with UnCertainties'.* In Mathematica User Conference, Champaign, Illinois, USA, 2008.
[37] avec D. Daney. *UnCertainties', a Package for Interval Analysis.* In Proceedings of the 9th International Mathematica Symposium, Maastricht, The Netherlands, 2008.

Chapitre 4

Un environnement logiciel pour la modélisation et la simulation

A quelques modifications près, les quatre premières sections de ce chapitre sont extraites d'un document de spécification pour Mosela, un environnement logiciel symbolique-numérique pour la modélisation et la simulation développé en collaboration avec Airbus depuis 2005. Il spécifie précisément les fonctionnalités que doit proposer un tel environnement. Partant de l'analyse des besoins industriels dans ce domaine, et de l'étude de l'existant, il définit ou précise les concepts et les méthodes sur lesquels doit s'appuyer le développement d'un tel environnement. Il propose aussi une structure et une architecture pour les différents éléments logiciels qui le composent. Enfin, il présente des choix sur la représentation des objets à manipuler et sur les algorithmes à implémenter dans ces différents composants.

La dernière section présente le travail qui a été effectivement réalisé, certaines difficultés rencontrées et les solutions développées pour les surmonter.

4.1 Présentation générale

On présente ici les motivations, les besoins en comparaison de l'existant, et l'architecture générale du projet.

4.1.1 Motivations

Dans un contexte industriel, l'activité de modélisation–simulation est l'apanage des bureaux d'études et constitue leur activité principale que ce soit au cours de phases de conception de systèmes complexes, de phases d'analyses des performances de ces systèmes une fois conçus ou de phases d'identification de paramètres du comportement de prototypes.

Le cycle de vie des modèles est une boucle où se succèdent une étape de développement du modèle, une étape de simulation, et enfin une étape de comparaison des résultats de la simulation avec des mesures qui conduit soit à la validation du modèle soit à un retour à l'étape de développement pour effectuer des modifications. Une fois le modèle validé, il sert à produire un code de simulation numérique efficace destinée à des simulations en temps réel ou à grande échelle.

Trois types d'acteurs collaborent et entrent en jeux successivement dans l'étape de développement de la boucle du cycle de vie des modèles :
- des ingénieurs "métiers" qui possèdent les connaissances et l'expérience du ou des domaines scientifiques concernés, et sont à même de faire les choix de modélisation, à savoir des paramètres et des formalismes pour représenter le système, et d'écrire les modèles sous forme équationnelle,
- des ingénieurs "méthodes" qui savent au mieux quelles méthodes numériques mettre en oeuvre pour expliciter ces équations et fabriquer, à partir de modèles les algorithmes de calculs efficaces et pertinents,
- des développeurs, chargés de l'implémentation, et donc du choix de la représentation informatique des données et des calculs, pour produire les codes de simulation numérique.

On souhaite permettre à ces différents acteurs de travailler au sein du même environnement logiciel et sur les mêmes objets et que le produit final de ce travail soit un code de simulation numérique efficace, satisfaisant aux contraintes d'écriture, de codage et de nommage en vigueur ainsi qu'à des contraintes de compatibilité éventuelles avec d'autres codes.

4.1.2 Besoins

Dans le cadre de ce projet, on s'attend à pouvoir dans cet environnement :
- définir un modèle, ou modifier un modèle existant,
- simuler facilement un modèle ou une partie de modèle pour le mettre au point, pour valider une hypothèse, ou une approximation
- générer le code de simulation numérique efficace correspondant
- valider numériquement le code de simulation généré

Définir un modèle A tout moment, en cours de définition, un modèle doit pouvoir être sauvegardé et/ou récupéré à partir d'une sauvegarde.

Un modèle est défini
- par la déclaration d'un certain nombre de variables auxquelles sont attachées diverses propriétés (grandeur physique, formalisme, unité, valeurs limites, etc...) ainsi que leur nature dans le modèle (entrée, sortie, intermédiaire) et leur caractéristique temporelle (constante ou dynamique)
- par la déclaration d'un certain nombre de relations entre ces variables sous forme analytique explicite et auxquelles sont également attachées diverses propriétés (domaine ou

hypothèse de validité, signification physique, etc...)
– par la déclaration d'un certain nombre de relation entre des variables et des variables d'autres modèles, de sorte que l'exécution de ces modèles permettraient le calcul de ces variables.

Un modèle est *valide* lorsque l'on peut en déduire un modèle calculatoire, i.e. une suite d'expressions permettant, à chaque pas de temps - et éventuellement après une phase d'initialisation - de calculer les sorties du modèle en fonction de ses entrées. La validité d'un modèle doit être vérifiable par l'environnement.

Simuler un modèle Lors de sa mise au point, on doit pouvoir rapidement vérifier certaines propriétés numériques d'un modèle, ou de certaines relations du modèle. L'environnement doit donc être capable de produire un code d'évaluation numérique interprétable et modifiable facilement, ainsi que posséder les fonctionnalités d'import de données et de visualisation de données nécessaires au calcul et à l'exploitation des sorties du modèle - en plus de fonctionnalités élémentaires de calcul numérique et de calcul symbolique.

Générer le code de simulation Une fois le modèle, et ses éventuels sous-modèles (modèles appelés au cours de l'évaluation) validés, on souhaite générer un code de simulation numérique efficace - par exemple un code C - satisfaisant certaines contraintes d'écriture. Il est important que ce code généré par l'environnement soit produit - et raisonnablement commenté - automatiquement et que son intégration dans un éventuel outil intégré de simulation puisse se faire sans la moindre intervention de l'utilisateur. Ainsi le code généré doit notamment gérer de manière autonome les allocations mémoires, les appels et interruptions ainsi que les erreurs éventuelles d'exécution.

Valider le code de simulation Pour valider par comparaison au code d'évaluation numérique, le code de simulation généré, ainsi que pour des raisons d'efficacité lors de la mise au point de gros modèles, on souhaite pouvoir appeler et exécuter depuis l'environnement le code de simulation, et ainsi effectuer des calculs de résultats à l'aide de celui-ci, et la visualisation des résultats obtenus aussi simplement qu'avec le code d'évaluation.

4.1.3 Existant

Deux des principaux outils généralistes de modélisation et de simulation actuellement utilisés est l'environnement Simulink* [Kar06] intégré dans l'environnement de calcul scienti-

∗. Simulink et Matlab sont deux produits MathWorks - www.mathworks.com/products

fique et technique Matlab * [Ett96], et son alternative libre Xcos † intégré dans l'environnement Scilab † .

Si l'on souhaite utiliser une approche analytique de la modélisation est d'utiliser le langage libre orienté-objet Modelica [Mat98] par l'intermédiaire d'un des environnements qui l'implémente, citons parmi les plus répandus Dymola ‡ , Xcos de nouveau † , MapleSim § , et MathModelica ¶, le premier étant intégré à Catia ‡ , la principale solution logicielle industrielle de conception de produits, les deux derniers reposant respectivement sur les deux principaux systèmes de calcul formel commerciaux, Maple § et Mathematica ‖.

Simulink C'est un environnement de modélisation et de simulation basé sur une approche graphique de la modélisation à l'aide de diagrammes interconnectant des blocs. Cette représentation est héritée de plusieurs générations de simulateurs/calculateurs analogiques, et s'appuie sur la notion de blocs qui sont définis par la nature de leurs entrées, de leurs sorties et de la transformation simple qui fournit les entrées en fonction des sorties (intégration, sommation, multiplication par une constante).

Même si les blocs Simulink sont infiniment plus riches que les blocs des calculateurs analogiques, cette méthode de description de systèmes reste limitée par le paradigme de calcul qu'elle implémente de fait, à base de variables d'état et d'équations différentielles ordinaires. Cette limite ne satisfait pas nos besoins en terme de description de modèles. De plus, les fonctionnalités de génération de code C de cette solution ne permettent pas facilement d'obtenir un code satisfaisant les contraintes de nommage et de codage imposées.

Modelica Apparu en 1997, le langage Modelica est le résultat d'un effort de standardisation pour faire converger de nombreuses expériences d'une approche orientée-objet de la modélisation. Un système est décrit par l'interconnexion de ses sous-systèmes. Chaque sous-système est défini comme une instance d'une classe prédéfinie d'objets de référence. C'est un objet auquel on sait attacher des variables et des équations. La diffusion de ce langage très complet a produit d'une part de nombreuses bibliothèques de composants, et d'autre part plusieurs implémentations d'analyseurs de cohérence de systèmes, de moteurs de simulation, et d'interfaces graphique de définitions de modèles, dont j'ai donné des exemples plus haut.

Quitte à redéfinir les fonctionnalités de génération de code C, et à retravailler les fonctionnalités de validation numérique, à la fois du modèle et du code C généré, on pourrait utiliser une solution existante à base de Modelica, mais devant l'ampleur de cette tâche, on préfèrera définir un langage de modélisation plus simple et implémenter seulement les

†. Les environnements Xcos et Scilab sont développés par le consortium Scilab - www.scilab.org
‡. Dymola et Catia sont deux produits Dassault System - www.3ds.com/fr/products/catia
§. MapleSim et Maple sont des produits MapleSoft - www.maplesoft.com/products
¶. MathModelica est un produit MathCore - www.mathcore.com/products/mathmodelica
‖. Mathematica est un produit Wolfram Research - www.wolfram.com

fonctionnalités nécessaires correspondantes à ce langage.

4.1.4 Architecture

La clef de voûte de l'architecture du projet est donc la définition d'un langage simple de modélisation - que nous désignerons dans la suite par *LSM* - qui soit capable de représenter nos besoins en termes de description de modèles.

On peut alors faire une analogie entre l'environnement que l'on souhaite développer et un environnement de développement logiciel. Ainsi notre environnement doit posséder quatre composants principaux qui s'appuient sur ce langage.
– un *éditeur* pour pouvoir fabriquer des modèles, i.e. des textes de ce langage,
– un *analyseur* syntaxique et lexical pour vérifier la validité des modèles,
– un *interprète* pour la fabrication du code d'évaluation numérique,
– un *compilateur* pour la génération du code C de simulation numérique.
La figure 4.1 montre comment s'organisent les composants en mettant en évidence deux perspectives, de conception et de validation.

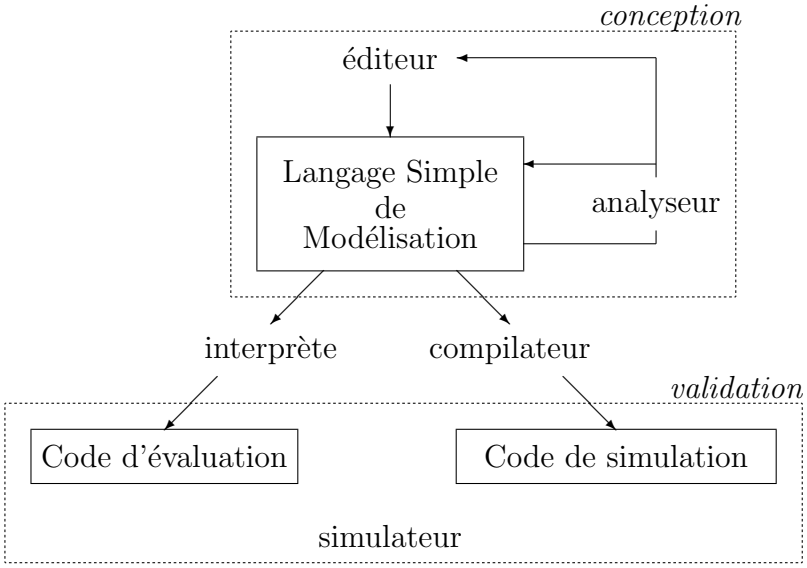

FIGURE 4.1 – Architecture de l'environnement

Il doit aussi intégrer comme cinquième composant un *simulateur* qui possède des fonctionnalités
– d'exécution numérique pour le code interprété (arithmétique en précision infinie, contrôle de la précision, arithmétique par intervalles, etc...),
– d'importation et d'exportation de données avec des formats variés depuis ou vers des bases de données,
– de communication avec des programmes de calcul externes,

– d'analyse et de visualisation de données.

Remarque 4.1 *Pour minimiser le travail de développement, on s'appuiera sur un logiciel de calcul formel car ce type de logiciel possède déjà l'essentiel des fonctionnalités demandées au simulateur, et possède des fonctionnalités de manipulation symbolique qui aideront à coder les quatre autres composants.*

4.2 Conception des modèles

Dans le cadre de ce projet, on s'intéresse à des modèles statiques ou constants - qui n'évoluent pas au cours du temps - et à des modèles dynamiques discrets ou continus discrétisés. Dans ces modèles, on distingue les constantes, des *variables dynamiques* qui seront recalculées à chaque instant. Le petit intervalle de temps qui sépare deux instants consécutifs et pendant lequel on suppose que les variables du modèle ne changent pas de valeur est appelé le *pas de temps*. La valeur d'une variable à un instant donné dépend de la valeur des constantes, de la valeur de certaines variables dynamiques à cet instant mais aussi éventuellement de valeurs de certaines variables à des instants précédents. Il faut donc distinguer le ou les premier(s) instant(s) de la simulation pour lesquels ces valeurs ne sont pas définies.

Lors de la simulation numérique de tels modèles, on distingue trois phases :
– le *chargement* du modèle pendant lequel sont calculées les valeurs des constantes,
– l'*initialisation* de la simulation qui calcule les valeurs des variables pour le(s) premier(s) instant(s) de la simulation,
– l'*exécution* de la simulation qui calcule les valeurs des variables pour chaque instant.
Et on doit déclarer, pour chaque variable dynamique comment elle se calcule lors de l'initialisation et lors de l'exécution. Par défaut, on considère que les deux calculs sont identiques si un seul est décrit.

On suppose de plus que ces modèles sont explicites, ou que les parties implicites ont été explicitées par des schémas de résolution. La description du modèle consiste à associer une variable à une expression analytique, ou à associer une liste de variables à un *programme de calcul*, précisément défini ci-dessous :

> Un *programme de calcul* est défini par la donnée
> – d'une liste de variables d'entrées,
> – d'une liste de variables de sorties
> – d'une liste d'instructions
> sachant que
> – la i-ème instruction est
> – soit une affectation à une variable temporaire de la valeur d'une expression analytique ne faisant intervenir que les variables d'entrées et les variables temporaires précédemment affectées,

  - soit une instruction de branchement, à laquelle sera substituée, en fonction du résultat d'un test ne faisant intervenir que les valeurs des variables d'entrées et des variables temporaires précédemment affectées, une nouvelle liste d'instructions,
  - soit une instruction de répétition, à laquelle sera substituée un certain nombre de fois une liste d'instructions, ce nombre dépendant de l'exécution d'une instruction de fin de répétition,
- toutes les variables de sorties sont affectées par des instructions au cours du programme, quelles que soient les branches des instructions de branchement considérées.

Ainsi la valeur de n'importe quelle variable à un instant donné s'obtient :
- s'il s'agit d'une entrée, par simple lecture ou requête à une base de donnée
- si elle est décrite par une expression analytique, par son évaluation,
- si elle est décrite par un programme de calcul, par l'exécution de ce programme de calcul,
- si elle est associée à une instance d'un autre modèle, par la simulation de ce sous-modèle.

Définition de fonctions

Pour éviter d'éventuelles répétitions dans les déclarations de variables, on peut définir des fonctions. Une fonction est associée soit à une expression analytique, soit à un programme de calcul, mais dans les deux cas, sa définition contient aussi une liste d'arguments qui interviennent dans ces expressions et qui seront passés par valeurs lors d'un appel de la fonction. Ces appels peuvent avoir lieu à l'intérieur de n'importe quelle expression analytique.

Remarque 4.2 *On peut éventuellement définir une fonction sans lui associer ni expression analytique, ni programme de calcul, mais en précisant la liste de ses arguments, la liste de ses sorties et les méthodes effectives qui permettent d'obtenir les sorties en fonctions des arguments dans les deux langages cibles de l'environnement, le code d'évaluation numérique et le code de simulation. Cette fonctionnalité sera particulièrement utile pour accéder, au cours de calculs, à des données tabulées par l'intermédiaire de fonctions d'interpolations.*

4.2.1 Langage Simple de Modélisation - LSM

Le langage Simple de Modélisation est un langage *déclaratif* qui va servir de base à la représentation des modèles dans tout l'environnement. On le définit ci-dessous par sa syntaxe abstraite dans un format à la XML :
- un document bien formé est constitué d'éléments,
- un élément est constitué d'un nom qui définit son type, d'un certain nombre d'attributs, et de son contenu,

– chaque attribut est une paire dont le premier composant est le nom de l'attribut et le second sa valeur,
– le contenu d'un élément est un ensemble d'éléments et de commentaires.

On peut passer de cette syntaxe abstraite à une syntaxe concrète classique à base de marqueurs (voir figure 4.2) ou à une syntaxe fonctionnelle avec des fonctions inertes (voir figure 4.3)

```
<element att1=val1 .. attN=valN>
      <element1>  ..  </element1>
       ..
      <elementP>  ..  </elementP>
</element>
```

FIGURE 4.2 – Syntaxe XML

```
Element[
  { Element1[..], ..,ElementN[..] },
     att1 -> val1,
     ..,
     attN -> valN ]
```

FIGURE 4.3 – Syntaxe fonctionnelle

Document

L'unité la plus englobante de description est le modèle qui se confond avec la notion de document. Dans ce langage, un modèle est décrit par un document bien formé qui contient à sa racine exactement un élément *modèle*.

Modèle Les attributs de l'élément *modèle* sont les suivants :
– *nom* pour le nom du modèle,
– *version* pour le numéro de version du modèle,
– *temps* pour préciser la variable qui représente le *pas-de-temps* le cas échéant,
– *sorties* pour préciser la liste des variables sorties du modèle.

Le contenu de l'élément *modèle* est un ensemble de déclarations, c'est à dire d'éléments *entrée*, *constante*, *relation*, *fonction* ou *sous-modèle*.

Déclarations

Entrée Les éléments *entrée* sont les déclarations des variables d'entrée du modèle. Les attributs de l'élément entrée sont :
- *nature* pour préciser si cette entrée est constante (à récupérer une fois lors du chargement du modèle) ou dynamique (à récupérer à chaque pas de temps) - on distingue alors le cas générique du cas d'initialisation de la simulation.
- *type* pour renseigner le type de la variable a déclarer lors de la génération de code. Les types simples sont booléens, entiers, flottants, et les types composés, vecteurs et matrices construits sur les types simples.
- *unité* pour l'unité physique de la grandeur associée (ce qui permettra des conversions automatiques si besoin pendant l'évaluation,
- *précision* permet d'attacher une précision à la variable,
- *incertitude* permet d'attacher un modèle d'incertitude à la variable

Le contenu de l'élément *entrée* est un élément *variable*.

Constante Les éléments *constante* sont les déclarations des constantes du modèle. Ils ont deux possibilités pour leur contenu,
- soit un élément *constante* est composé de :
  - un élément *variable* définissant la constante déclarée,
  - un élément *expression_analytique* exprimant la valeur de la constante déclarée,
- soit il est composé de :
  - un élément *liste_de_variables* définissant les constantes déclarées,
  - un élément *programme_de_calcul* ayant pour sorties les constantes déclarées.

Ils ont comme attribut le même attribut *unité* que l'élément *entrée* dont la valeur selon le cas est une liste.

Relation Les éléments *relation* sont les déclarations des variables dynamiques du modèle. Ils ont plusieurs attributs :
- un attribut booléen *initialisation* qui précise que la déclaration concerne l'initialisation du modèle ou l'exécution du modèle,
- un attribut *validité* qui précise quels sont les domaines de validité pour les variables qui interviennent dans la déclaration,
- un attribut *incertitude* qui permet d'attacher un modèle d'incertitude de méthode au calcul de la variable déclarée,
- et le même attribut *unité* que l'élément *constante*.

Comme un élément *constante*, l'élément *relation* a deux formes possibles pour son contenu :
- soit il est composé de :
  - un élément *variable* définissant la variable déclarée,
  - un élément *expression_analytique* exprimant la valeur de la variable déclarée,
- soit il est composé de :

– un élément *liste_de_variables* définissant les variables déclarées,
– un élément *programme_de_calcul* ayant pour sorties les variables déclarées.

Fonction Les éléments *fonction* sont utilisés pour déclarer les fonctions utilisées dans le modèle. Ils ont un attribut booléen *externe* qui est vrai lorsque la fonction est définie ni par une expression analytique, ni par un programme de calcul, dans ce cas, les moyens de traduire cette fonction sont respectivement stockés dans les attributs *code_évaluation* et *code_simulation*.

Ils ont pour contenu :
– un *symbole* représentant le nom de la fonction,
– un élément *liste_de_variables* définissant les arguments de la fonction,
– un élément *variable* ou *liste_de_variables* définissant la ou les sortie(s) de la fonction,
– un élément *expression_analytique* ou *programme_de_calcul* sauf dans le cas de fonction *externe*.

Sous-modèle Un élément *sous_modèle* déclare les variables qui sont calculées par la simulation d'un autre modèle. Il a pour attributs :
– *nom* pour le nom du modèle appelé,
– *version* pour la version du modèle appelé,
– *phase* pour préciser dans quelle phase (chargement, initialisation, ou exécution) le modèle est appelé
– *instant* pour préciser à quel instant le modèle est appelé (dans le cas d'un modèle appelé en exécution. Cet attribut permet d'envisager le multiplexage entre modèles, c'est à dire de faire tourner le modèle appelé plus rapidement que le modèle appelant.

Le contenu d'un élément *sous-modèle* est
– un élément *liste_de_variables* qui déclare les variables du modèle appelant qui vont être fournies en entrée au modèle appelé,
– un élément *liste_de_variables* qui déclare les entrées correspondantes du modèle appelé,
– un élément *liste_de_variables* qui déclare les variables du modèle appelant qui vont être récupérées en sortie du modèle appelé,
– un élément *liste_de_variables* qui déclare les variables du modèle appelé correspondantes.

Eléments de base

Plus bas dans l'arbre de syntaxe abstraite du langage, on trouve encore des éléments qui représentent des objets structurés complexes ainsi que des objets simples.

Les objets simples considérés comme les éléments terminaux du langage sont :
– les *nombres* (entiers et flottants),
– les *symboles*,
– et les *chaînes_de_caractères*.

A partir de ces objets simples on construit l'éléments de base *variable*, puis *liste_de_variables* et *expression_analytique* et enfin *programme_de_calcul*.

Variable Un élément *variable* a comme contenu un *symbole*. Il a comme attributs :
- *type* qui précise le type de la variable à déclarer lors de la génération du code de simulation (booléen, entier, flottant, vecteur, matrice)
- *instant* pour caractériser les variables calculées à des instants précédant l'instant courant de simulation.

Liste de variables Cet élément qui ne sert qu'à regrouper des *variables* n'a pas d'attribut et son contenu est un ensemble de *variables*.

Expression analytique L'élément *expression_analytique* n'a pas d'attribut, et son contenu est une expression analytique faisant intervenir des *variables* au sens du système de calcul formel dans lequel on va réaliser l'implémentation de l'environnement. On pourrait considérer une *expression_analytique* comme un élément terminal du langage, mais ce n'est pas précis car c'est un élément construit sur des *variables* et non sur n'importe quel *symbole*. Pour être plus rigoureux, on pourrait donner une définition récursive terminale d'*expression_analytique* décrivant son contenu comme
- soit un élément *variable*
- soit un *symbole* et un ensemble d'éléments *expression_analytique* pour représenter l'application d'un opérateur :

  $$\text{symbole}(\text{expr}_1, \ldots, \text{expr}_n)$$

  avec la contrainte sémantique que le *symbole* soit un opérateur connu de l'environnement ou défini comme une *fonction* dans le langage.

Remarque 4.3 *Lors de l'implémentation, on évitera l'alourdissement syntaxique du à cette représentation récursive, en l'implicitant dès le premier niveau de récursion.*

Programme de calcul Un élément *programme_de_calcul* n'a pas d'attribut. Il a pour contenu :
- un élément *liste_de_variables* précisant ses variables d'entrée,
- un élément *liste_de_variables* précisant ses variables de sortie,
- un ensemble d'éléments *instruction*.

Instruction Un élément *instruction* a un attribut *nature* qui précise si il s'agit d'une affection, d'une instruction de branchement, d'une instruction de répétition, ou d'un bloc d'instructions. Il a pour contenu selon le cas :
- s'il s'agit d'une affectation :

  - un élément *variable*,
  - un élément *expression_analytique*,
- s'il s'agit d'une instruction de branchement :
  - un élément *expression_analytique* pour exprimer la condition,
  - un élément *instruction* à exécuter dans le cas où la condition est vraie,
  - un élément *instruction* à exécuter dans le cas où elle est fausse,
- s'il s'agit d'une instruction de répétition :
  - un élément *expression_analytique* pour exprimer la condition de fin de répétition,
  - un élément *instruction* à exécuter en boucle,
- s'il s'agit d'un bloc d'instructions :
  - un ensemble d'éléments *instruction*.

4.3 Analyse de modèles

L'analyse de modèle a pour objet d'étudier le bon fondement sémantique d'un modèle syntaxiquement valide, et d'en extraire les informations nécessaires soit au concepteur au cours de l'édition du modèle, soit une fois le modèle terminé, à l'interprète ou au compilateur pour la production d'un code d'évaluation ou de simulation.

La première étape est l'analyse lexicale, puis syntaxique qui à partir d'un texte du Langage Simple de Modélisation représentant un modèle construit la représentation de ce modèle par exemple sous la forme d'un arbre, chaque élément étant à la racine d'un sous-arbre représentant son contenu, et en supposant que chaque noeud est étiqueté par le type de l'élément représenté et que l'on dispose d'un accès indépendant aux attributs pour chaque élément. C'est la description de la grammaire de syntaxe abstraite présentée dans la section précédente qui permet de fabriquer systématiquement, sans plus d'information, un analyseur lexical et un analyseur syntaxique. On veillera à connecter ces analyseurs à l'éditeur pour qu'en cas d'erreurs, ce dernier puisse les localiser automatiquement.

La plupart des méthodes d'analyse sont alors des parcours d'arbre au cours desquels un contexte est mis à jour, la sortie calculée par la méthode se déduisant de ce contexte en fin de parcours.

4.3.1 Validation sémantique

La validation sémantique est aussi une étape préliminaire à toute autre analyse, et doit pouvoir être réalisée à n'importe quel moment, au cours de l'édition.

On va décrire la validation sémantique de chaque type d'élément, éventuellement en terme de validation sémantique de son contenu, ce qui induit implicitement le parcours d'arbre en profondeur d'abord. Cette méthode maintient à jour dans un contexte les informations suivantes :

– le modèle courant (nom et version),
– la liste des variables,
– la liste des constantes,
– la liste des variables dynamiques,
– la liste des variables dynamiques définies à l'initialisation de la simulation,
– la liste des variables dynamiques définies génériquement,
– la liste des fonctions,
– la liste des variables locales,
– la liste des sorties.

Modèle La validation sémantique du *modèle* procède à l'initialisation du contexte, définit le modèle courant à partir des attributs *nom* et *version* - ou sa valeur par défaut -, ajoute la variable de l'attribut *temps* - ou sa valeur par défaut - à la liste des variables et à la liste des constantes, positionne la liste des sorties à partir de l'attribut *sorties*. Elle reporte une erreur si l'attribut *nom* ou l'attribut *sorties* n'a pas de valeur. Puis, elle lance la validation sémantique itérativement sur les déclarations, éléments de son contenu. Enfin, elle vérifie que la liste des sorties est incluse dans la liste des variables, reporte une erreur sinon.

Entrée La validation sémantique d'un élément *entrée* considère l'élément *variable* de son contenu et vérifie que son attribut *instant* n'est pas positionné (i.e la variable est considérée au pas de temps courant). En fonction de son attribut *nature*, elle ajoute la variable aux listes de variables du contexte correspondantes. Elle reporte une erreur si cette même variable appartient déjà à une de ces listes (seule une variable dynamique peut-être déclarée deux fois, une fois dans le cas générique, une fois dans le cas d'initialisation de la simulation).

Constante La validation sémantique d'un élément *constante* considère son contenu, et selon le cas,
– valide sémantiquement son élément *expression_analytique*, puis considère l'élément *variable* de son contenu et vérifie que son attribut *instant* n'est pas positionné, enfin ajoute la variable à la liste des constantes et à la liste des variables - elle reporte une erreur si cette variable appartient déjà à une de ces deux listes.
– valide sémantiquement son élément *programme_de_calcul*, puis considère dans son contenu l'élément *liste_de_variables* et, pour chacune des *variable* du contenu de cet élément, vérifie que son attribut *instant* n'est pas positionné, et ajoute la variable à la liste des constantes et à la liste des variables - elle reporte une erreur si cette variable appartient déjà à une de ces deux listes. Enfin, elle vérifie que la liste de variables, sorties du *programme_de_calcul* est bien incluse dans l'élément *liste_de_variables* de la *constante*.

Relation La validation sémantique d'un élément *relation* considère son contenu, et selon le cas,

- valide sémantiquement son élément *expression_analytique*, puis considère l'élément *variable* de son contenu et vérifie que son attribut *instant* n'est pas positionné, enfin ajoute la variable à la liste des variables dynamiques et à la liste des variables, et en fonction de l'attribut *initialisation*, à la liste des variables définies à l'initialisation, ou à la liste des variables définies génériquement - elle reporte une erreur si la variable appartient déjà à la liste correspondante, ou à la liste des constantes.
- valide sémantiquement son élément *programme_de_calcul*, puis considère dans son contenu l'élément *liste_de_variables* et, pour chacune des *variable* du contenu de cet élément, procède comme expliqué ci-dessus, en reportant une erreur dans les mêmes cas. Enfin, elle vérifie que la liste de variables, sorties du *programme_de_calcul* est bien incluse dans l'élément *liste_de_variables* de la *relation*.

Fonction La validation sémantique d'un élément *fonction* vérifie que

- soit l'attribut *externe* est positionné et dans ce cas, les attributs *code_évaluation* et *code_simulation* le sont aussi, et l'élément ne contient pas d'élément *expression_analytique* ni *programme_de_calcul*
- ou l'inverse, dans ce cas, elle vérifie aussi que le contenu est bien constitué d'une *variable* pour la sortie et d'une *expression_analytique* ou d'une *liste_de_variables* et d'un *programme_de_calcul*. Puis elle procéde à la vérification sémantique de l'*expression_analytique* ou du *programme_de_calcul*.

Dans les deux cas, les variables de la liste des arguments de la fonction sont ajoutées à la liste des variables locales.

Sous-modèle La validation sémantique d'un élément *sous-modèle* considère son contenu et vérifie la cohérence entre les longueurs des différentes listes de variables. En fonction de la valeur de l'attribut *phase*, elle ajoute les variables reçues du sous-modèles aux listes de variables correspondantes du contexte en vérifiant qu'elles n'ont pas déjà été déclarées, et elle vérifie qu'il n'y a pas d'incompatibilité entre la nature induite des variables passées au sous-modèle et ces mêmes listes.

Remarque 4.4 *Optionnellement, la validation sémantique peut maintenir dans son contexte l'ensemble des modèles connus de l'environnement avec pour chaque modèle l'ensemble des versions connues, elle peut alors vérifier que les attributs* nom *et* version *correspondent à un modèle et à une version connus.*

Expression analytique La validation sémantique des expressions analytiques consiste ici essentiellement à vérifier que les opérateurs utilisés sont bien des opérateurs connus ou des fonctions du modèle et à vérifier la cohérence entre les opérateurs et leurs opérandes,

notamment en ce qui concerne leur nombre. Elle sera déléguée au système de calcul formel dans lequel on va réaliser l'environnement.

Programme de calcul et instructions La validation sémantique d'un élément *programme_de_calcul* consiste à valider sémantiquement ses instructions, et à vérifier si chacune de ses variables de sortie apparait effectivement au moins une fois comme *variable* dans une instruction d'affectation. Les variables qui apparaissent comme *variable* dans une instruction d'affectation mais ne sont pas des variables de sorties sont ajoutées à la liste des variables locales.

Un élément *instruction* est valide sémantiquement lorsque
- s'il s'agit d'une affectation : l'expression analytique est valide,
- s'il s'agit d'une instruction de branchement :
  - l'*expression_analytique* pour exprimer la condition est valide
  - les *instructions* à exécuter dans les deux cas sont valides et contiennent des instructions d'affectation qui définissent les mêmes variables,
- s'il s'agit d'une instruction de répétition : l' *expression_analytique* pour exprimer la condition de fin de répétition, et l'élément *instruction* à exécuter en boucle sont valides,
- s'il s'agit d'un bloc d'instructions : toutes ces *instructions* sont valides.

4.3.2 Variables, dépendances et complétude

On a vu comment la validation sémantique d'un modèle maintient, pour des raisons internes, différentes listes de variables en fonction de leurs natures. Ces listes, si elles sont proposées à l'utilisateur pendant l'édition d'un modèle peuvent s'avérer une aide précieuse, surtout lorsque les modèles sont gros et qu'il devient difficile de mémoriser toutes les variables déjà introduites. Par construction, il existe des relations d'inclusions entre ces listes qui sont représentées dans la figure 4.4. On considère par défaut que si une variable dynamique n'a pas de définition spéciale pour le cas d'initialisation, la définition pour le cas générique lui est associée.

Ainsi, on pourra, soit faire en sorte que le contexte maintenu par la validation persiste et soit disponible pour l'interface d'édition, soit coder un parcours d'arbre simplifié qui maintient un complexe similaire, sans s'occuper de validation, et est exécuté systématiquement comme préliminaire à toute analyse du modèle.

Au cours de l'édition, il est intéressant pour l'utilisateur de savoir quelles sont, en l'état actuel du modèle, les variables qu'il utilise sans qu'elles aient été déclarées - ou définies - ou quelles sont les variables qui ont été déclarées mais ne sont pas utilisées. Pour cela, on est amené à calculer les dépendances entre variables, localement pour chaque déclaration et à propager ces dépendances.

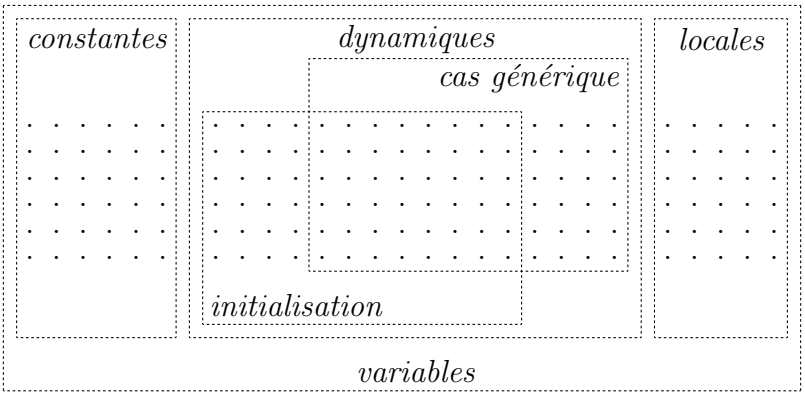

FIGURE 4.4 – Listes de variables

Variables

Une première analyse sur les variables consiste à extraire toutes les variables apparaissant dans le modèle, et à comparer l'ensemble obtenu à la réunion de la liste de toutes les variables (globales) et de la liste des variables locales. Si il existe des variables dans le premier ensemble qui n'appartiennent pas au deuxième, cela traduit que certaines expressions du modèle font référence à des variables qui ne sont pas définies. L'égalité de ces ensembles par contre ne garantit pas que l'on puisse exécuter le modèle, une analyse de dépendance et de complétude doit être réalisée pour cela, et elle doit être réalisée pour les trois phases d'utilisation du modèle.

Dépendances

On analyse un modèle pour calculer les dépendances entre variables en extrayant du modèle toutes les déclarations et en les analysant itérativement. Pour chaque déclaration on représente son impact en terme de dépendance entre variables sous la forme d'une règle qui, à l'ensemble des variables qui doivent être connues pour "effectuer" la déclaration associe le complémentaire de cet ensemble à l'ensemble des variables qui seront connues après. Nous allons voir précisément comment cela se fait sur chaque type de déclaration.

Remarque 4.5 *Un traitement spécial est réservé aux déclarations de fonction qui ne sont pas considérées dans cette analyse. A la place, on développera chaque appel de fonction rencontré au cours de l'analyse des autres déclarations dans une expression analytique ou dans un programme de calcul, et cela récursivement.*

Les résultats sont stockés dans trois listes qui correspondent aux trois phases d'utilisation du modèle :

– la liste des règles de dépendance entre constantes,
– la liste des règles de dépendance pour l'initialisation du modèle,
– la liste des règles de dépendance pour l'exécution du modèle.
Les éléments de type *entrée* ne sont pas considérés dans cet analyse, car leur traitement est trivial. On suppose que l'on dispose par ailleurs de la liste des variables d'entrées - supposées comme connues lors de la simulation - , déclinée en liste des entrées constantes, liste des entrées pour l'initialisation, et liste des entrées pour l'exécution.

Constante L'analyse des éléments de type *constante* fournit des règles de dépendance pour la phase constante : selon le cas,
– on va associer à l'ensemble des variables apparaissant dans l'*expression_analytique* la *variable* de son contenu,
– ou alors on va associer à l'ensemble des variables d'entrée du *programme_de_calcul*, la *liste_de_variables* de son contenu.

Relation Une analyse similaire des éléments de type *relation* produit des règles de même nature que l'on ajoute, selon l'attribut *initialisation* soit aux règles de dépendance pour l'initialisation, soit aux règles de dépendance pour l'exécution.

Sous-modèle La valeur de l'attribut *phase* de l'élément *sous-modéle* détermine la liste à laquelle est ajoutée la règle de dépendance qui se déduit directement de la *liste_de_variables* fournie au sous-modèle et de la *liste_de_variables* récupérée du sous-modèle.

Remarque 4.6 *Il est important de noter qu'au cours de cette analyse de dépendance, on ne "rentre" pas à l'intérieur des éléments* programme_de_calcul *et l'on considère que chaque variable de sortie dépends de toutes les variables d'entrée. Ceci ne correspond pas forcément à la réalité physique, mais correspond bien à la réalité calculatoire. On fait le même choix pour les appels de sous-modèles.*

Remarque 4.7 *On notera aussi que, lors de l'analyse de dépendances pour l'exécution du modèle, on ne considère pas les dépendances faisant intervenir des variables définies à des instants précédant l'instant courant. En effet, toutes ces variables ont été calculées à des instants de simulation précédents et leurs valeurs sont donc connues.*

Remarque 4.8 *Il faut noter enfin qu'en fin de traitement, la liste des règles de dépendances pour l'initialisation du modèle est complétée par certaines règles de la liste des règles de dépendances pour l'exécution du modèle pour prendre en compte l'hypothèse par défaut pour les variables dynamiques qui n'ont pas de déclaration spécifique pour la phase d'initialisation.*

Complétude

Une fois les dépendances locales analysées, on applique, pour chaque phase d'utilisation du modèle, un algorithme de propagation (voir algorithme 1) à partir des listes de variables d'entrées qui doit retourner l'ensemble de toutes les variables :
- dans le cas constant, on prend en entrée la liste des entrées constantes et l'on compare l'ensemble obtenu à la liste des variables constantes,
- pour la phase d'initialisation, on prend en entrée la réunion de la liste des constantes et de liste des entrées dynamiques pour l'initialisation et l'on compare l'ensemble obtenu à la liste des variables dynamiques,
- pour la phase d'exécution, on prend en entrée la réunion de la liste des constantes et de liste des entrées dynamiques pour l'exécution et l'on compare l'ensemble obtenu à la liste des variables dynamiques.

On obtient ainsi la garantie de pouvoir calculer toutes les variables ou, par différence, la liste des variables que l'on ne saura pas calculer.

Algorithme 1 : Propage les dépendances à partir des entrées

Input : $\mathcal{E}$ et $\mathcal{R}$, respectivement l'ensemble des variables d'entrées et l'ensemble des règles de dépendances
Output : $\mathcal{V}$, l'ensemble des variables que le modèle permet de calculer

initialisation;
$\mathcal{V} \leftarrow \mathcal{E}$;
point fixe;
$\mathcal{V}_{prev} \leftarrow \mathcal{V}$;
repeat *Propage*
  foreach *règle* $(\mathcal{V}_{in} \mapsto \mathcal{V}_{out}) \in \mathcal{R}$ **do**
    if $(\mathcal{V}_{in} \in \mathcal{V})$ **then**
      $\mathcal{V} \longleftarrow (\mathcal{V} \cup \mathcal{V}_{out})$
until $\mathcal{V} = \mathcal{V}_{prev}$;
return $\mathcal{V}$;

Un algorithme de propagation dans l'autre sens à partir des sorties du modèle (voir algorithme 2) permet de déceler des boucles et éventuellement de repérer des variables qui sont définies mais ne sont pas utilisées. Il faut, pour déceler les boucles, garder une trace, au cours de l'exécution de la boucle "Propage", des valeurs successives de l'ensemble $\mathcal{V}$ des variables que l'on ne sait pas encore calculer et interrompre l'algorithme si la valeur courante de $\mathcal{V}$ est une valeur qu'il a déjà prise. Pour repérer les variables définies mais non utilisées, il faut garder une trace des règles utilisées au cours de l'exécution de l'algorithme et les comparer à l'ensemble de toutes les règles.

Là encore, on applique cet algorithme pour chaque phase d'utilisation du modèle, et à chaque fois avec les ensembles de variables et les listes de règles correspondants.

Algorithme 2 : Propage les dépendances depuis les sorties

Input : $\mathcal{E}$, $\mathcal{S}$ et $\mathcal{R}$, respectivement l'ensemble des variables d'entrée, de sortie et l'ensemble des règles de dépendances

Output : $\mathcal{V}$, l'ensemble des variables que l'on ne sait pas encore calculer

initialisation;
$\mathcal{V} \leftarrow (\mathcal{S} \setminus \mathcal{E})$;
repeat *Propage*
  choisir $v \in \mathcal{V}$;
  trouver une règle $(\mathcal{V}_{in} \mapsto \mathcal{V}_{out}) \in \mathcal{R}$ telle que $v \in \mathcal{V}_{out}$;
  $\mathcal{V} \leftarrow ((\mathcal{V} \setminus \mathcal{V}_{out}) \cup \mathcal{V}_{in}) \setminus \mathcal{E}$;
until $\mathcal{V} = \emptyset$;
return $\mathcal{V}$;

4.3.3 Calculs de grandeurs et d'unités

On a vu la possibilité d'attacher à une *entrée* une unité physique. A travers cette unité, on récupère deux informations,
– une information de grandeur - par exemple si l'unité est le *micron*, on sait que la variable représente une *longueur*,
– et une information d'échelle, qui peut s'apprécier relativement à l'unité choisie par le Système International - sur l'exemple du *micron*, ce sera 10^{-6}

L'analyse d'un modèle doit permettre de vérifier la cohérence des grandeurs à travers les relations décrites par le modèle et la cohérence des échelles. Pour ce faire, on va générer un code d'évaluation du modèle utilisant une arithmétique exotique implémentant le calcul des unités et l'exécuter.

Si l'on utilise un système de calcul formel pour implémenter l'environnement, il n'est pas nécessaire d'ordonner les déclarations de ce modèle pour l'évaluer. En effet, ce genre de système possède généralement un mécanisme d'évaluation *retardée* - ou "jusqu'au bout", qui permet de suivre la chaîne des affectations, indépendamment de l'ordre d'apparition de ces affections. Ainsi, si on évalue successivement les affectations

$$\begin{aligned} a &\longleftarrow b \\ b &\longleftarrow 1 \end{aligned}$$

l'évaluation de a donne 1.

On s'intéressera aux problèmes d'ordonnancement de calcul dans le cadre de la génération de code de simulation, dans la section 4.4.1.

Arithmétique sur les grandeurs

Les grandeurs physiques de base - et leurs unités associées sont :

– la longueur - mètre
– la masse - kilogramme
– le temps - seconde
– le courant électrique - ampère
– la température - kelvin
– la quantité de matière - mole
– l'intensité lumineuse - candela

A partir de ces sept grandeurs de base, on dérive l'ensemble des grandeurs physiques** multiplicativement, en voici quelques exemples :

– la fréquence - hertz $\longrightarrow$ temps^{-1}
– l'accélération $\longrightarrow$ longueur.temps^{-2}
– la force - newton $\longrightarrow$ masse.longueur.temps^{-2}
– la pression - pascal $\longrightarrow$ masse.longueur^{-1}.temps^{-2}
– la puissance - watt $\longrightarrow$ masse.longueur^{2}.temps^{-3}
– etc...

Ainsi, au prix d'une conversion préliminaire des grandeurs dérivées en grandeur de base, on peut se limiter aux grandeurs de base. On définit une arithmétique sur les grandeurs en conservant la définition habituelle de la multiplication et en implémentant les règles suivantes pour l'addition :

$$x + y \mapsto \begin{cases} x \text{ si } x = y \\ \text{erreur sinon} \end{cases}$$

L'évaluation du modèle dans cette arithmétique à partir des grandeurs dérivées des unités des entrées permet d'associer des grandeurs aux autres variables du modèle ou éventuellement de vérifier la cohérence de celles déjà associées. Elle permet aussi de repérer les erreurs dans les expressions concernant des incohérences d'unités.

Arithmétique sur les unités

Les conversions d'unités sont fréquentes dans les modèles industrielles. Parfois explicites, elles peuvent aussi être cachées par une constante numérique qui apparaît non dimensionnée dans une expression analytique. Il peut être intéressant pour l'utilisateur de mettre en évidence ces conversions, et cela est possible par une simple évaluation du modèle en considérant un nouveau type de valeurs que l'on peut donner aux variables. Cette évaluation est aussi une alternative à l'implémentation de l'arithmétique sur les grandeurs du paragraphe précédent

Considérons chaque valeur comme un monôme dans l'un des sept indéterminées suivantes `longueur`, `masse`, `temps`, `courant`, `temperature`, `matiere`, `lumiere`, en acceptant des puissances entières négatives pour pouvoir représenter n'importe quelle grandeur. Le calcul

**. voir www.industrie.gouv.fr/metro/aquoisert/si.htm pour un document de référence sur le Système International d'unités, ou le décret n° 61-501 du 3 mai 1961

de chaque variable, par le biais d'une expression analytique ou d'un programme de calcul retourne
– soit le même genre de monôme, et dans ce cas, c'est dans le coefficient que l'on trouvera de l'information sur d'éventuelles conversions d'échelles,
– soit une expression faisant intervenir des sommes, ou des appels de fonctions numériques sur des indéterminées, et dans ce cas démontrera une incohérence des grandeurs.

4.4 Compilation

A l'issue de l'analyse, le compilateur produit, selon deux méthodes différentes,
– un *code d'évaluation* pour la mise au point rapide du modèle
– et un *code de simulation* numérique.
Les qualités du premier sont la lisibilité, l'interopérabilité avec les autres fonctionnalités de l'environnement héritées du système de calcul formel, - graphiques pour la visualisation, arithmétiques en précision infinie ou d'intervalles pour étudier les qualités numériques du modèle (précision, fiabilité, exactitude), l'interactivité avec les données nécessaires à ces études, et la facilité d'utilisation.

Les qualités du second sont l'efficacité et son interopérabilité avec l'environnement industriel de simulation. Ce code généré, doit être précisément, aux noms de fonctions et de variable près, aux allocations de mémoires près, identique à celui que l'industriel aurait codé pour simuler le même modèle.

Code d'évaluation

Une simple traduction des déclarations d'éléments en des définitions selon le cas, de symboles, de fonctions ou de procédures, permet de fabriquer ce code d'évaluation, étant données les capacités d'évaluation des systèmes de calcul formel dont on a déjà parlé dans la section précédente.

On utilisera simplement un symbole pour représenter une constante, ou une entrée constante. On utilisera une syntaxe fonctionnelle $v(t)$ pour représenter une variable intervenant au cours de la phase d'exécution du modèle et la même syntaxe correspondant à l'instant initial $v(t_0)$ pour la même variable intervenant au cours de la phase d'intialisation du modèle.

Les points techniques à régler pour implémenter cette traduction dépendent beaucoup de la syntaxe et plus généralement du système de calcul formel utilisé. Leur description sort du cadre de ce document.

Code de simulation

En ce qui concerne le code de simulation, dont le langage cible est forcément, ne serait-ce que pour des raisons d'efficacité un code de bas niveau, sa génération nécessite d'abord
- de calculer le type des variables intervenant dans le modèle pour gérer l'allocation de mémoire
- de calculer l'ordre dans lequel on doit évaluer les affectations traduisant les déclarations du modèle, puisque que l'on ne pourra utiliser une variable dans un calcul que si on lui a auparavant affecté une valeur.

4.4.1 Ordonnancement

Cette phase d'ordonnancement du calcul s'exécute après l'analyse du modèle, on suppose que celui-ci est complet et cohérent et donc qu'il permet le calcul des variables de sortie à partir des variables d'entrée.

Pour chacune des phases d'utilisation du modèle, on va ordonner les déclarations correspondantes, en fonctions des dépendances entre les variables qu'elles définissent :

- Soit $\{d_1, d_2, \ldots, d_n\}$ la liste des déclarations avant ordonnancement,
- soit

$$r_1 = (\mathcal{V}_{\text{in}}^1 \longrightarrow \mathcal{V}_{\text{out}}^1), r_2 = (\mathcal{V}_{\text{in}}^2 \longrightarrow \mathcal{V}_{\text{out}}^2), \ldots, r_n = (\mathcal{V}_{\text{in}}^n \longrightarrow \mathcal{V}_{\text{out}}^n)$$

  la liste des règles de dépendance correspondante,
- soit $\left\{d_{\alpha(1)}, d_{\alpha(2)}, \ldots, d_{\alpha(n)}\right\}$ la liste des déclarations après ordonnancement,
- alors $\alpha(i) < \alpha(j)$, c'est à dire que la i-ème déclaration se trouve avant la j-ème dans la liste ordonnée seulement si aucune variable d'entrée de la i-ème déclaration n'est une variable de sortie de la j-ème :

$$\mathcal{V}_{\text{in}}^{\alpha(i)} \cap \mathcal{V}_{\text{out}}^{\alpha(j)} = \emptyset$$

Un algorithme d'ordonnancement consiste à procéder par comparaison et à itérer la recherche du premier élément de la liste à trier (voir algorithme 3).

C'est l'absence de boucle dans le graphe de dépendance du modèle - que l'on a vérifiée pendant l'analyse - qui garantit la bonne terminaison de cet algorithme. On pourrait aussi, en mémorisant les valeurs successives de $\mathcal{L}$ repérer une boucle et s'en servir pour déceler une boucle dans le graphe de dépendance.

4.4.2 Typage dynamique

Le typage de toutes les variables du modèle est réalisé par la propagation des types des variables d'entrée et des types des variables qui interviennent dans le modèle à des instants différents, après ordonnancement.

Algorithme 3 : ordonnancement

Input : $\mathcal{L} = \{(d_1, \mathcal{V}^1_{\text{in}} \longrightarrow \mathcal{V}^1_{\text{out}}), (d_2, \mathcal{V}^2_{\text{in}} \longrightarrow \mathcal{V}^2_{\text{out}}), \ldots, (d_n, \mathcal{V}^n_{\text{in}} \longrightarrow \mathcal{V}^n_{\text{out}})\}$ la liste des déclarations et de leurs règles de dépendances avant ordonnancement

Output : $\mathcal{L}_{\text{out}} = \left\{d_{\alpha(1)}, d_{\alpha(2)}, \ldots, d_{\alpha(n)}\right\}$ la liste des déclarations après ordonnancement

repeat *Place*
  $(d, v_{\text{in}}, v_{\text{out}}) \longleftarrow \mathsf{first}(\mathcal{L})$;
  $\mathcal{L} \longleftarrow \mathsf{rest}(\mathcal{L})$;
  if $(v_{\text{in}} \cap (\cup_{\mathcal{L}} \mathcal{V}_{\text{out}})) = \emptyset$ **then**
    $\mathcal{L}_{\text{out}} \longleftarrow \mathsf{append}(\mathcal{L}_{\text{out}}, d)$;
  else
    $\mathcal{L} \longleftarrow \mathsf{append}(\mathcal{L}, (d, v_{\text{in}}, v_{\text{out}}))$
until $\mathcal{L} = \emptyset$;
return $\mathcal{L}_{\text{out}}$;

Remarque 4.9 *Il est important que l'utilisateur fournisse les types des variables qui interviennent dans le modèle à des instants différents car on n'en a pas tenu compte dans le calcul du graphe de dépendance (voir remarque 4.7). Le choix alternatif aurait conduit dans la plupart des cas à des boucles dans le graphe de dépendance.*

La méthode de typage est en tout point similaire au calcul réalisé pour la détermination et la vérification de la cohérence des unités : il faut définir une arithmétique différente sur les types en fonction du langage informatique cible - par exemple C.

Les types *chaîne_de_caractère* et *booléen* ne sont pas utilisés dans les variables, on considère seulement les types numériques *entier* et *flottant*, et les types complexes *vecteur* et *matrice* construits sur l'un des types numériques, ces types complexes comportent explicitement leur(s) dimension(s).

Pour les opérateurs de somme et de produit, par exemple, on a les tables suivantes :

$+$	E	F	$V_E(n)$	$V_F(n)$	$M_E(l,c)$	$M_F(l,c)$
E	E	F	X	X	X	X
F	F	F	X	X	X	X
$V_E(n)$	X	X	$V_E(n)$	$V_F(n)$	X	X
$V_E(\neq n)$	X	X	X	X	X	X
$V_F(n)$	X	X	$V_F(n)$	$V_F(n)$	X	X
$V_F(\neq n)$	X	X	X	X	X	X
$M_E(l,c)$	X	X	X	X	$M_E(l,c)$	$M_F(l,c)$
$M_E(\neq (l,c))$	X	X	X	X	X	X
$M_F(l,c)$	X	X	X	X	$M_F(l,c)$	$M_F(l,c)$
$M_F(\neq (l,c))$	X	X	X	X	X	X

.	E	F	$V_E(n)$	$V_F(n)$	$M_E(l,c)$	$M_F(l,c)$
E	E	F	$V_E(n)$	$V_F(n)$	$M_E(l,c)$	$M_F(l,c)$
F	F	F	$V_F(n)$	$V_F(n)$	$M_F(l,c)$	$M_F(l,c)$
$V_E(n)$	$V_E(n)$	$V_F(n)$	E	F	X	X
$V_E(l)$	$V_E(l)$	$V_F(l)$	X	X	$V_E(c)$	$V_F(c)$
$V_F(n)$	$V_F(n)$	$V_F(n)$	E	F	X	X
$V_F(l)$	$V_F(l)$	$V_F(l)$	X	X	$V_F(c)$	$V_F(c)$
$M_E(l,n)$	$M_E(l,n)$	$M_F(l,n)$	$V_E(l)$	$V_F(l)$	X	X
$M_E(p,l)$	$M_E(p,l)$	$M_F(p,l)$	X	X	$M_E(p,c)$	$M_F(p,c)$
$M_E(p,q)$	$M_E(p,q)$	$M_F(p,q)$	X	X	X	X
$M_F(l,n)$	$M_F(l,n)$	$M_F(l,n)$	$V_F(l)$	$V_F(l)$	X	X
$M_F(p,l)$	$M_F(p,l)$	$M_F(p,l)$	X	X	$M_F(p,c)$	$M_F(p,c)$
$M_F(p,q)$	$M_F(p,q)$	$M_F(p,q)$	X	X	X	X

Chaque X dans la table correspond à une incompatibilité de type dans une opération qui doit retourner une erreur. La propagation des types dans tous les calculs du modèle donne un type à toutes les variables du modèle et permet notamment de vérifier la cohérence avec les types déclarés par l'utilisateur. Une fois les types déterminés, ils sont stockés comme attribut dans les éléments *variable*.

4.4.3 Génération de code

La manière dont l'environnement va générer le code dépend fortement des règles et des contraintes de nommage et de codage à respecter.

Dans le cas du C, la génération du code concerne aussi bien les fichiers de code source que les fichiers d'entête. Dans certains fichiers, peu de morceaux de code dépendent du modèle, alors que dans d'autres, peu de morceaux de codes restent inchangés d'un modèle à l'autre. La génération de code C combine donc deux méthodes :
– l'instanciation de modèle de code (*template*),
– la génération de code à partir de la traduction d'expression et de la représentation formelle du squelette du code source à l'aide de macros.
Les systèmes de calcul formel possèdent en général de bonnes fonctionnalités de traduction d'expression analytique en C. Certains offrent de plus la possibilité de représenter par des objets spécifiques les éléments de la syntaxe abstraite du langage C [††]

Le processus de génération de code est alors le suivant :
– en fonction des contraintes de codage, définir,
  – le squelette du programme de simulation complet, implémentant les 3 phases de chargement, d'initialisation et d'exécution, incluant des séquences d'évaluation des déclarations du modèle, et gérant les variables globales.

††. SymbolicC à partir de la version 8 de Mathematica, MacroC pour Maple

  - les représentations des programmes d'acquisition des données d'entrées à partir de l'environnement de simulation,
  - une représentation générique d'un programme de calcul d'une variable à partir d'une expression analytique,
  - une représentation générique d'un programme de calcul d'une liste de variables à partir d'un programme de calcul,
- instancier, pour chaque déclaration du modèle, une des deux représentations génériques précédentes et générer le code source des programmes correspondants,
- définir les séquences d'évaluation des déclarations du modèle, sous la forme d'appels de ces programmes - en tenant compte de l'ordonnancement calculé en début de compilation -
- incorporer ces séquences à la représentation du programme de simulation et générer le ou les fichiers de code source correspondant.

4.5 Réalisations

L'environnement de modélisation et de simulation Mosela que j'ai développé en suivant des spécifications proches de celles exposées ci-dessus est aujourd'hui un prototype pré-industriel complètement opérationnel appartenant à Airbus. Il est constitué d'un ensemble de 34 bibliothèques de code source Mathematica réparties thématiquement (représentation des objets et fondamentales, édition de modèles, analyse de modèles, génération de code C, génération de code d'évaluation, génération de documentation, simulation numérique) pour un total de près de 14.200 lignes de code Mathematica.

Testé sur des dizaines de modèles élaborés au bureau d'étude Airbus, il est à même de traiter très efficacement des modèles de plusieurs centaines de variables ou faisant appels à plusieurs instances de plusieurs sous-modèles. Pour chacun de ces modèles, il est capable de produire en quelques (fractions de) secondes :
- un code C compatible avec les normes de codage Airbus et donc prêt à être intégré aux outils internes de simulation de vol,
- la documentation métier des modèles dans un format très similaire à l'existant,
- un code Mathematica d'évaluation numérique du modèle compatible avec une arithmétique par intervalles.

Il contient aussi un environnement de simulation numérique capable d'importer les données numériques métiers fournies par les bases de données du bureau d'étude, de simuler le comportement du modèle à partir de son code Mathematica d'évaluation ou de son code C de simulation, et permettant la visualisation des résultats obtenus.

Une des principales difficultés rencontrées lors du développement de ce prototype concerne le typage automatique des variables et les vérifications de cohérence des types dans les expressions symbolique. Même avec un choix relativement restreint de types de bases (entiers, flottants, flottants en double précision) et la possibilité de ne construire sur ces types

que des vecteurs ou des matrices, la propagation du typage à travers un modèle après ordonnancement - voir section 4.4.2 - s'est avérée difficile à implémenter et à mettre au point, notamment en présence d'appels de fonctions faisant intervenir des programmes de calculs, et en présence d'instructions de branchement multiples. Une adaptation de cette méthode mais qui prend en compte les déclarations de types des variables en sortie, et ne considère qu'une des branches à chaque instruction de branchement a été développée et intégrée au prototype. Un calcul supplémentaire de cohérence des types est alors nécessaire. Une méthode alternative, basée sur la programmation par contraintes, et qui ne nécessite aucun ordonnancement préalable a aussi été étudiée : on définit un univers des possibles où chaque variable peut être de n'importe quel type, et que l'on filtre ensuite par les contraintes induites par chacune des déclarations du modèle.

Perspectives

Pour conclure ce mémoire et la synthèse de mes travaux de recherche, j'aimerais revenir sur une réflexion qui a fait l'objet d'une communication il y a quelques mois[43] .

J'ai commencé à utiliser des systèmes de calcul formel à une époque où leur principale originalité était leur mécanisme de représentation et d'évaluation des objets leur permettant notamment de considérer des symboles sans les associer nécessairement à des valeurs numériques et d'autre part de faire des calculs exacts sur les entiers et sur les rationnels.

Quand l'essentiel d'une communauté scientifique a vu l'extraordinaire potentiel de ces outils pour étudier les systèmes algébriques et a aidé à leur développement dans ce sens, en traduisant calcul formel par *Computer Algebra*, j'ai préféré chercher à exploiter leur potentiel de représentation et de manipulation d'objets symboliques, basé sur une vision très universelle et très homogène à base d'arbres, bien adaptée à la programmation fonctionnelle et à l'application de règles de transformation, en traduisant alors calcul formel par *Symbolic Computation*.

Aujourd'hui les principales évolutions commerciales de ces systèmes, les logiciels *Maple* et *Mathematica*, sont devenus des environnements de calcul, voire de résolution de problèmes très complets et très performants. Ils ont intégrés les algorithmes les plus performants de manipulation et de résolution algébrique. Ils se sont ouverts au calcul numérique en développant des bibliothèques d'analyse numérique, de calcul matriciel qui n'ont rien à envier en termes de performances à celles des logiciels spécialisés. Ils ont étendu leur capacité de représentation et de manipulation symbolique aux graphiques, aux images, au son, aux textes, aux interfaces interactives pour certain, tout en conservant cet homogénéité de représentation arborescente qui les rend si intuitifs et si performants pour l'expérimentation et le prototypage.

Toutefois, et alors que *Matlab*, leur principal concurrent en tant qu'environnement de calcul et de résolution de problème, s'imposait dans les bureaux d'études et le monde industriel, avec l'aide de sa solution intégrée de modélisation et de simulation *Simulink*, ils n'ont pas jusqu'à présent exploité leur potentiel de manipulation symbolique dans ce domaine. Certes, ces dernières années ont vu apparaître les initiatives *MapleSim* et *MathModelica* qui permettent à *Maple* et à *Mathematica* d'enrichir leur fonctionnalités en communicant avec ces environnements de modélisation/simulation. Mais en choisissant de s'appuyer sur le forma-

lisme un peu lourd - quoiqu'extraordinairement universel et efficace - du langage *Modelica*, aucun de ces deux logiciels n'a voulu intégrer dans son propre environnement les *modèles*, simplement en tant qu'objet symbolique, avec la même homogénéité de représentation.

L'enjeu de cette intégration serait d'avoir un environnement de résolution de problème vraiment universel, un outil de prototypage et d'expérimentation absolu. La contrepartie tient en quelques points à implémenter, à quelques concepts à modifier, quelques manipulations à simplifier, décrits ci-dessous.

Gérer un contexte La modélisation est, comme on l'a vu, une activité descriptive, déclarative. Actuellement, chaque ligne de commande évalué produit des calculs et en effet de bord la modification du contexte de l'évaluateur où sont gardées les informations et les définitions concernant une session d'utilisation. Cette notion de *contexte* devrait être étendu et un système devrait pouvoir gérer simultanément et efficacement plusieurs contextes de déclarations qui correspondraient à une représentation légère et naturelle des modèles.

Attacher des propriétés aux symboles Les variables, et leurs représentations, sont centrales dans l'activité de modélisation et de simulation. Représentées par des symboles dans un système de calcul formel, on devrait pouvoir leur attacher naturellement des propriétés et faire en sorte que certains calculs réalisés avec ces variables propagent ou vérifient la cohérence des propriétés qui leur sont attachées. Un exemple de traitement similaire est implémenté sur les valeurs numériques dans un de ces systèmes qui attache une notion de précision aux variables et propage les précisions des variables à travers les calculs. On voudrait la possibilité de définir ainsi les propriétés et les mécanismes de propagation, simplement et efficacement.

Manipuler des programmes Dans les modèles, les programmes de calculs sont une extension naturelle et indispensable des expressions algébriques. Même si ces programmes, et notamment leur expression la plus simple, les Straight Line Program (SLP) ont été l'objet de nombreuses études théoriques, on peut s'étonner que les systèmes de calcul formel ne les manipulent, ni les considèrent aujourd'hui avec autant d'aisance que les expressions analytiques, sur le plan de l'évaluation, la composition, la différentiation, etc.... C'est une extension indispensable pour la modélisation.

Evaluer de façon réversible Entre modélisation et simulation, on voudrait pouvoir aller alternativement d'une forme évaluée des expressions (pour leur évaluation numérique) à une forme non évaluée (pour leur analyse symbolique) et conserver l'accès naturellement à ces deux formes au cours des calculs. Ainsi à l'évaluation immédiate, et à l'évaluation retardée, il faudrait ajouter une évaluation réversible pour revenir au modèle après la simulation.

Références

[Moo65] Ramon E. Moore. *Interval Analysis.* Prentice-Hall, Inc. Englewood Cliffs, New Jersey, 1965.

[Elm78] H. Elmqvist. *A Structured Model Langage for Large Continuous Sysytems.* PhD thesis, Report CODEN :LUTFD2/(TFRT 1015), Dept. of Automatic Control, Lund Institute of Technology, Lund, Sweden, 1978.

[Bro83] Rodney A. Brooks and Tomas Lozano-Perez. *A subdivision algorithm in configuration space for findpath with rotation.* In Proceedings of the Eighth international joint conference on Artificial intelligence - Volume 2 (IJCAI'83), pp. 799-806. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA. 1983

[Gom90] C. Gomez *MACROFORT : a Fortran code generator in MAPLE* INRIA, RT-0119, 1990.

[Mer93] J-P. Merlet. *Les Robots Parallèles.* Hermès, Paris. 1990

[Laz92] D. Lazard. *Stewart platform and Gröbner basis.* In ARK, pages 136-142, Ferrare, September, 7-9, 1992.

[Abb95] J. Abbott, A. Van Leeuwen and A. Strotmann *Objectives of OpenMath* J. Symbolic Computation, vol.11, 1995.

[Ett96] D. M. Etter *Introduction to MATLAB for Engineers and Scientists* Prentice Hall, 1996.

[Phil97] S. Philoreau. *Contribution du Calcul Formel à l'Ingénierie Optique : Elaboration d'une méthode de gestion des performances d'une charge utile optique.* PhD thesis, INIST-CNRS T114274, 1997.

[Mat98] S. E. Mattsson and H. Elmqvist. *An Overview of the Modeling Langage Modelica.* EuroSim'98 Simulation Congress, April 1998, Helsinki.

[Mou98] B. Mourrain. *Computing isolated polynomial roots by matrix methods.* J. of Symbolic Computation, vol.26, no.6, pages 715-738, 1998.

[Mer02] J-P. Merlet. *ALIAS, a System Solving Library Based on Interval Analysis* ERCIM News, vol.50, 2002.

[Kar06] Steven T. Karris. *Introduction to Simulink with Engineering Applications*. Orchard Publications, 978-0974423975, 2006.

Publications

Ouvrages

[O1] Y. Papegay. *Outils formels pour la modélisation en mécanique*. PhD thesis, Université de Nice Sophia-Antipolis, France, November 1992.

[O2] Y. Papegay, editor. *Applied Mathematica, Proc. of the 8th Int. Mathematica Symposium*. INRIA, Avignon, Juin 2006.

Articles de journaux

[A3] S. Dalmas, Y. Papegay, and H. Prieto. Mathematica as an openmath application. *SIGSAM Bulletin*, 2000.

[A4] Y. Papegay, J-P. Merlet, and D. Daney. Exact kinematics analysis of car's suspension mechanisms using symbolic computation and interval analysis. *Mechanism and Machine Theory*. 40, 395-413, 2005.

[A5] D. Daney, B. Madeline, and Y. Papegay. Choosing Measurement Poses for Robot Calibration with Local Convergence Method and Tabu Search. *International Journal of Robotic Research*, 24(6) :501–518, 2005.

[A6] D. Daney, N. Andreff, G. Chabert, and Y. Papegay. Interval method for calibration of parallel robots : a vision-based experimentation. *Mechanism and Machine Theory*, 41(8) :929–944, August 2006.

[A7] Y. Papegay. Exploring board game strategies. *The Mathematica Journal*, 10(2) :363–395, 2006.

[A8] Yves Papegay, editor. *IMS Special Issue*, volume 11 Issue 2 and 3 of *The Mathematica Journal*. Wolfram Media, 2009.

Conférences internationales avec comité de lecture

[C9] C. Garnier, Y. Papegay, and P. Rideau. Modélisation dynamique littérale. *Journal of Computer Methods in Applied Mechanics and Engineering*, 75 :215–225, 1989.

[C10] H. Hirukawa, Y. Papegay, and T. Matsui. A motion planning algorithm for convex polyhedra in contact under translation and rotation. In *proceedings of the 1994 IEEE International Conference on Robotics and Automation*, pages 3020–3028, San Diego, California, May 1994. IEEE Robotics and Automation Society.

[C11] H. Hirukawa, Y. Papegay, and H. Tsukune. A motion planning algorithm of polyhedra in contact for mechanical assembly. In *proceedings of the 20th International Conference on Industrial Electronics Control and Instrumentation*, pages 924–929, Bologna, Italia, September 1994. IEEE Industrial Electronics Society.

[C12] H. Hirukawa and Y. Papegay. A lazy algorithm for planning motions in contact. In *proceedings of the 1994 IEEE/RSJ/GI International Conference on Intelligent Robots and Systems*, pages 2152–2159, Munich, Germany, September 1994. IEEE Industrial Electronics Society.

[C13] Y. Papegay and S. Philoreau. Design and performances analysis of complex optical devices using symbolic computation. In Scientists Inc., editor, *Proceedings of the 2nd ASCM*, pages 55–61, Tokyo, August 1996. H. Kobayashi Ed.

[C14] Y. Papegay and S. Dalmas. Quaterman vs. quaternica : a comparative implementation of quaternions in maple and mathematica. In Scientists Inc., editor, *Proceedings of the 2nd ASCM*, pages 113–120, Tokyo, August 1996. H. Kobayashi Ed.

[C15] Y. Papegay and W. Klein. Automatic generation of a mos transistor simulation model with maple v. *Nuclear Instruments and Methods in Physics Research A*, 389 :125–127, 1997.

[C16] H. Hirukawa and Y. Papegay. Motion planning of objects in contact by the silhouette algorithm. In *proceedings of the 2000 IEEE International Conference on Robotics and Automation*, pages 3020–3028, San Francisco, California, May 2000. IEEE Robotics and Automation Society.

[C17] H. Hirukawa, B.Mourrain, and Y. Papegay. A symbolic-numeric silhouette algorithm. In *Proceedings of 2000 IEEE International conference on RObotics Systems*.

[C18] Y. Papegay, D. Daney, and J.P. Merlet. Parallel implementation of interval analysis for equations solving. In *EuroPVM/MPI 2003 conference*, Venice, 2003. published in Lecture Notes in Computer Sciences 2840,Springer Verlag.

[C19] D. Daney, Y. Papegay, and A. Neumaier. Interval methods for certification of the kinematic calibration of parallel robots. In *Proc. IEEE International Conference on Robotics and Automation (ICRA)*, pages 147–152, New Orleans, USA, April 28-30 2004.

[C20] D. Daney, A. Andreff, and Y. Papegay. Interval method for calibration of parallel robots : a vision-based experimentation. In *Computational Kinematics*, Cassino, 2005.

[C21] C. Grandon, D. Daney, Y. Papegay, C. Tavolieri, E. Ottaviano, and M. Ceccarelli. Handling Uncertainties with Symbolic/Numerical Solvers for a Class of Parallel Robots. In *WC - International Federation for the Theory of Machines and Mechanisms (IFToMM)*, June 2007.

[C22] G. Trombettoni, Y. Papegay, G. Chabert, and O. Pourtallier. A Box-Consistency Contraction Operator Based on Extremal Functions. In *Abstract in the GAMM/IMACS International Symposium on Scientific Computing, Computer Arithmetic and Verified Numerical Computations (SCAN)*, El Paso, TX,USA, 2008.

[C23] G. Trombettoni, Y. Papegay, G. Chabert, and O. Pourtallier. A Box-Consistency Contractor Based on Extremal Functions. In *Proc. CP, Constraint Programming, LNCS 6308*, pages 491–498. Springer, 2010.

Autres conférences internationales

[Ci24] Y. Papegay. Le calcul formel : un outil pour l'ingenieur.(exposé invité) In Universitat Politecnica de Catalunya, editor, *TEMU-95*, February 1995.

[Ci25] Y. Papegay and S. Dalmas. Quaternica : A package for manipulating expressions involving quaternions. In Computational mechanics publications, editor, *Mathematics with vision : proceedings of the First International Mathematica Symposium*, pages 291–298, 1995.

[Ci26] Y. Papegay and S. Dalmas. Design and performance analysis of optical instruments with computer algebra. In University of New Mexico, editor, *Electronic proceedings of the first IMACS conference on Applications of Computer Algebra*, 1995. http ://math.unm.edu/ACA/Proceedings/MainPage.html.

[Ci27] S. Philoreau, D. Miras, Y. Papegay, and D. Simeoni. Circe : A new approach to performance management of optical instruments. In SPIE, editor, *Proceedings of the SPIE 96 Annual Meeting, volume 2817*. Infrared Spaceborne Remote Sensing Conference, August 1996.

[Ci28] Y. Papegay. From modeling to simulation with symbolic computation : An application to design and performance analysis of complex optical devices. In *Proceedings of the Second Workshop on Computer Algebra in Scientific Computing*. Munich, June 1999. Springer, Telos, ISBN-354066047X.

[Ci29] Y. Papegay. From maple 9 worksheets to mathematica notebooks : An almost automatic converter. In *Electronic Proceedings of the Wolfram 2004 Technology Conference*, Urbana Champaign, Illinois, USA, October 2004.

[Ci30] Y. Papegay and D. Daney. Solving with interval analysis. In *Wolfram Technology Conference*, Illinois, USA, 2005.

[Ci31] Y. Papegay. Exploring boardgame strategies. In *Proceedings of the 7th International Mathematica Symposium*, Perth, Western Australia, 2005.

[Ci32] C. Grandón, D. Daney, and Y. Papegay. Combining CP and interval methods for solving the direct kinematic of a parallel robot under uncertainties. In *Workshop of the Int. Conf. on Principles and Practice of Constraint Programming (IntCP 06)*, September 2006.

[Ci33] L. Farenc and Y. Papegay. Modeling flight dynamics for real-time simulator applications. In *Applied Mathematica, Proc. of the 8th Int. Mathematica Symposium*, Juin 2006.

[Ci34] Y. Papegay. Industrial C code generation, SymbolicC : a C wrapper. In *Wolfram Technology Conference*, Champaign, Illinois, USA, 2006.

[Ci35] Y. Papegay and D. Daney. Computing with Intervals : the Ultimate Symbolic Computation ? In *Wolfram Technology Conference*, Champaign, Illinois, USA, 2007.

[Ci36] , Y. Papegay, Certified Computations with UnCertainties‘, In *Mathematica User Conference*, Champaign, Illinois, USA, 2008.

[Ci37] , Y. Papegay and D. Daney. UnCertainties, a Package for Interval Analysis, In *Proceedings of the 9th International Mathematica Symposium*, Maastricht, The Netherlands, 2008.

[Ci38] , Y. Papegay and O. Pourtallier, Exploring Routing Strategies for a Virtual Yacht Race, In *Proceedings of the 9th International Mathematica Symposium*, Maastricht, The Netherlands, 2008.

[Ci39] N. Ramdani and Y. Papegay. Guaranteed Numerical Integration of Nonlinear Parametric ODEs. In *Proceedings of the 9th International Mathematica Symposium*, Maastricht, The Netherlands, 2008.

[Ci40] Y. Papegay. Delegating Computations to C Code. In *International Mathematica User Conference*, Champaign, Illinois, USA, 2009.

[Ci41] Y. Papegay. An application of modelling and simulation tools to elections. In *10th International Mathematica Symposium*, Beijing, China, 2010.

[Ci42] Y. Papegay. Modeling and simulation of french elections. In *Wolfram Technology Conference*, Champaign, Illinois, USA, 2010.

[Ci43] Y. Papegay. A formal approach for modeling and simulation. In *Wolfram Technology Conference*, Champaign, Illinois, USA, 2011.

Conférences nationales

[Cn44] Y. Papegay. Calcul formel pour la modélisation en mécanique. In *CALSYF, 9*. GRECO de Calcul Formel, 1991.

[Cn45] Y. Papegay. Modélisation et simulation avec Mathematica. October 2004. Conférence Mathematica Paris 2004.

[Cn46] Y. Papegay. Génération de code C avec Mathematica : le packetage SymbolicC. In *Conférence Mathematica Paris 2007*, Juin 2007.

[Cn47] Y. Papegay. Garantir le calcul numérique malgré les incertitudes. In *Conférence Mathematica Paris 2010*, Juin 2010.

Rapports techniques

[Rt48] C. Garnier, P. Rideau, and Y. Papegay. Modélisation dynamique littérale. Rapport technique 1255 CA/TC, aerospatiale, March 1987.

[Rt49] Y. Papegay, P. Capolsini, L. Pottier, and P. Rideau. Outils d'aide à la modélisation formelle en mécanique et en automatique. Rapport Technique CA/TSV 173, aerospatiale/INRIA, December 1991.

[Rt50] Patrick Capolsini, Stéphane Dalmas, and Yves Papegay. Quaterman : une bibliothèque *maple* de calculs et de manipulations sur l'algèbre des quaternions - a library for computing with quaternions in Maple. Rapport technique 148, INRIA-I3S, jan 1993.

[Rt51] Y. Papegay. Un logiciel d'aide à la conception et à l'évaluation des performances d'instruments optiques. Rapport Technique, aerospatiale/INRIA, février 1995.

[Rt52] Y. Papegay. Aide à la conception et à l'évaluation des performances d'instruments optiques. Rapport Technique, aerospatiale/INRIA, mars 1996.

[Rt53] B. Mourrain and Y. Papegay. Ecole de printemps de calcul formel. Support de Cours C-172, INRIA, Avril 1997.

[Rt54] C. Faure and Y. Papegay. Odyssee version 1.6 – the user's reference manual. Rapport technique 211, INRIA, 1997.

[Rt55] C. Faure and Y. Papegay. Odyssee version 1.7 – the user's reference manual. Rapport technique 224, INRIA, 1997.

[Rt56] Y. Papegay. ypama : un assistant à la modélisation. Rapport d'étude, INRIA/Airbus France, Janvier 2002.

[Rt57] Y. Papegay. ypama : un assistant à la modélisation, comment implémenter la désorientation des modèles. Rapport d'étude, INRIA/Airbus France, December 2003.

[Rt58] Y. Papegay. ypama : un assistant à la modélisation, analyse dimensionnelle et cohérence des unités. Rapport d'étude, INRIA/Airbus France, April 2004.

[Rt59] Y. Papegay. The Mosela Modeling and Simulation Environment. Rapport d'étude, INRIA/Airbus France, Janvier 2007.

[Rt60] Y. Papegay. The Mosela Modeling and Simulation Environment : Code Generation. Rapport d'étude, INRIA/Airbus France, Avril 2008.

[Rt61] Y. Papegay. The Mosela Modeling and Simulation Environment : Model Verification. Rapport d'etude, INRIA/Airbus France, Avril 2009.

[Rt62] Y. Papegay. The Mosela Modeling and Simulation Environment : Efficient Compilation. Rapport d'etude, INRIA/Airbus France, December 2010.

[Rt63] Y. Papegay. The Mosela Modeling and Simulation Environment : User's Guide. Rapport d'etude, INRIA/Airbus France, November 2011.

Curriculum Vitae

Notice individuelle

– Yves PAPEGAY
– Né le 5 décembre 1963 à NICE
– Français
– Pacs, 2 enfants
– Adresse :

  La Pinède A, 450 avenue Saint Philippe, 06410 BIOT

– Adresse professionnelle :

  Institut National de Recherche en Informatique et en Automatique,
  EPI COPRIN, B.P. 93, 06902 SOPHIA ANTIPOLIS Cedex
  Tél : 04 92 38 76 99

– courriel :

  yves.papegay@inria.fr

Titres

– Maîtrise de Mathématiques Pures (1984 – Université de Nice)
– C.A.P.E.S en Mathématiques (1985)
– D.E.A. de Mathématiques, option Calcul Formel (1986 – Université de Nice)
– Doctorat mention Sciences, spécialité Sciences Pour l'Ingénieur
  – titre de la thèse :
    "Outils Formels pour la Modélisation en Mécanique"
  – directeur de thèse :
    J. Morgenstern, Professeur d'informatique à l'Université de Nice Sophia-Antipolis

Curriculum Vitae

- de septembre 1980 à juin 1984 :

  étudiant à l'Université de Nice (DEUG, Licence, Maîtrise)
- de septembre 1984 à juin 1985 :

  professeur stagiaire au C.P.R. d'Aix-Marseille
- de septembre 1985 à septembre 1986 :

  étudiant à l'Université de Nice (DEA)
- de septembre 1986 à fevrier 1993 :

  étudiant en thèse de Doctorat (UNSA / INRIA Sophia Antipolis)
- de décembre 1987 à novembre 1988 :

  Service National, Base Aérienne 943
- de mars 1993 à novembre 1993 :

  chercheur à ElectroTechnical Laboratory (Tsukuba, Japon)
  boursier post-doctoral "Science and Technology Agency of Japan" (M.I.T.I.)
- depuis décembre 1993 :

  chargé de recherches à l'INRIA Sophia Antipolis
  - 2ème classe titulaire en juin 1995
  - 1ère classe en janvier 2000

Autres activités

- enseignement

  en moyenne 40 heures par an de 1993 à 2009 dans les disciplines et enseignements suivants :
  Info-Maple en classes préparatoires(Centre International de Valbonne), Licence Technicom (Université de Polynésie Française), Maîtrise Math-Info (Université de Nice Sophia Antipolis), DEA (UNSA, ENS)
- encadrements de stages
  - environ un stagiaire par an depuis 1993 (Maîtrise Math-Info, ESSI, Ecole Nationale Ingénieur de Rabat, MATMECA Bordeaux)
  - co-encadrements de thèses (J.Hubert, C.Tavolieri, J.Alexandre-dit-Sandretto)
- président du comité de programme de l'International Mathematica Symposium 2006
- membre de la commission de spécialiste 4 de l'Université de Polynésie Française de 2004 à 2008.