Étude de la complexité des logiciels de type flots de données en vue de la fiabilité : application à l'atelier logiciel SAGA

Marcel Chevalier

THESE

présentée à

l'Université Scientifique et Médicale de Grenoble

pour obtenir le grade de
DOCTEUR DE L'UNIVERSITE
JOSEPH FOURIER - GRENOBLE I
(Arrêté ministériel du 5 juillet 1984)
« Informatique »

par

Marcel CHEVALIER

ETUDE DE LA COMPLEXITE DES LOGICIELS DE TYPE "FLOTS DE DONNEES" EN VUE DE LA FIABILITE ; APPLICATION A L'ATELIER LOGICIEL SAGA

Thèse soutenue le 28 novembre 1989 devant la Commission d'Examen :

J. FONLUPT	Président
J.L. SOLER	Directeur
J.C. LAPRIE	Rapporteur
P. CASPI	Rapporteur
J.L. BERGERAND	Examinateur

Thèse préparée au sein du laboratoire I.M.A.G. - T.I.M. 3

UNIVERSITE Joseph FOURIER (GRENOBLE I)

Président de l'Université :
M. NEMOZ Alain

Année Universitaire 1988 - 1989

MEMBRES DU CORPS ENSEIGNANT DE SCIENCES ET DE GEOGRAPHIE

PROFESSEURS DE 1ère Classe

ADIBA Michel	Informatique
ANTOINE Pierre	Géologie I.R.I.G.M.
ARNAUD Paul	Chimie Organique
ARVIEU Robert	Physique Nucléaire I.S.N.
AUBERT Guy	Physique C.N.R.S
AURIAULT Jean-Louis	Mécanique
AYANT Yves	Physique Approfondie
BARBIER Marie-Jeanne	Electrochimie
BARJON Robert	Physique Nucléaire ISN
BARNOUD Fernand	Biochimie Macromoléculaire Végétale
BARRA Jean-René	Statistiques-Mathématiques Appliquées
BECKER Pierre	Physique
BEGUIN Claude	Chimie Organique
BELORISKY Elie	Physique
BENZAKEN Claude	Mathématiques Pures
BERARD Pierre	Mathématiques Pures
BERNARD Alain	Mathématiques Pures
BERTRANDIAS Françoise	Mathématiques Pures
BERTRANDIAS Jean-Paul	Mathématiques Pures
BILLET Jean	Géographie
BOELHER Jean-Paul	Mécanique
BRAVARD Yves	Géographie
CARLIER Georges	Biologie Végétale
CASTAING Bernard	Physique
CAUQUIS Georges	Chimie Organique
CHARDON Michel	Géographie
CHIBON Pierre	Biologie Animale
COHEN ADDAD Jean-Pierre	Physique
COLIN DE VERDIERE Yves	Mathématiques Pures
CYROT Michel	Physique du Solide
DEBELMAS Jacques	Géologie Générale
DEGRANGE Charles	Zoologie
DEMAILLY Jean-Pierre	Mathématiques Pures
DENEUVILLE Alain	Physique
DEPORTES Charles	Chimie Minérale
DOLIQUE Jean-Michel	Physique des Plasmas
DOUCE Roland	Physiologie Végétale
DUCROS Pierre	Cristallographie
FINKE Gerde	Informatique
GAGNAIRE Didier	Chimie Physique
GAUTRON René	Chimie
GENIES Eugène	Chimie
GERMAIN Jean-Pierre	Mécanique,
GIDON Maurice	Géologie
GUITTON Jacques	Chimie
HICTER Pierre	Chimie
IDELMAN Simon	Physiologie Animale
JANIN Bernard	Géographie
JOLY Jean René	Mathématiques Pures

JOSELEAU Jean Paul	Biochimie
KAHANE André, détaché	Physique
KAHANE Josette	Physique
KRAKOWIAK Sacha	Mathématiques Appliquées
LAJZEROWICZ Jeanine	Physique
LAJZEROWICZ Joseph	Physique
LAURENT Pierre-Jean	Mathématiques Appliquées
LEBRETON Alain	Mathématiques Appliquées
DE LEIRIS Joël	Biologie
LHOMME Jean	Chimie
LLIBOUTRY Louis	Géophysique
LOISEAUX Jean-Marie	Sciences Nucléaires I.S.N.
LONGEQUEUE Nicole	Physique
LUNA Domingo	Mathématiques Pures
MACHE Régis	Physiologie Végétale
MASCLE Georges	Géologie
MAYNARD Roger	Physique du Solide
OMONT Alain	Astrophysique
OZENDA Paul	Botanique (Biologie Végétale)
PANNETIER Jean	Chimie
PAYAN Jean-Jacques	Mathématiques Pures
PEBAY-PEYROULA Jean-Claude	Physique
PERRIER Guy	Géophysique
PIERRE Jean Louis	Chimie Organique
RENARD Michel	Thermodynamique
RIEDTMANN Christine	Mathématiques
RINAUDO Marguerite	Chimie CERMAV
ROSSI André	Biologie
SAXOD Raymond	Biologie Animale
SENGEL Philippe	Biologie Animale
SERGERAERT Francis	Mathématiques Pures
SOUCHIER Bernard	Biologie
SOUTIF Michel	Physique
STUTZ Pierre	Mécanique
TRILLING Laurent	Mathématiques Appliquées
VAN CUTSEM Bernard	Mathématiques Appliquées
VIALON Pierre	Géologie

PROFESSEURS de 2ème Classe

ARMAND Gilbert	Géographie
ATTANE Pierre	Mécanique
BARET Paul	Chimie
BERTIN José	Mathématiques
BLANCHI J.Pierre	STAPS
BLOCK Marc	Biologie
BLUM Jacques	Mathématiques Appliquées
BOITET Christian	Mathématiques Appliquées
BORNAREL Jean	Physique
BORRIONE Dominique	Automatique informatique
BOUVET Jean	Biologie
BROSSARD Jean	Mathématiques
BRUANDET J.François	Physique
BRUGAL Gérard	Biologie
BRUN Gilbert	Biologie
CASTAING Bernard	Physique
CERFF Rudiger	Biologie
CHIARAMELLA Yves	Mathématiques Appliquées
CHOLLET Jean Pierre	Mécanique
COLOMBEAU Jean François	Mathématiques (ENSL)
COURT Jean	Chimie
CUNIN Pierre Yves	Informatique
DAVID Jean	Géographie

DHOUAILLY Danielle	Biologie
DUFRESNOY Alain	Mathématiques Pures
GASPARD François	Physique
GIDON Maurice	Géologie
GIGNOUX Claude	Sciences Nucléaires
GILLARD Roland	Mathématiques Pures
GIORNI Alain	Sciences Nucléaires
GONZALEZ SPRINBERG Gérardo	Mathématiques Pures
GUIGO Maryse	Géographie
GUMUCHAIN Hervé	Géographie
HACQUES Gérard	Mathématiques Appliquées
HERBIN Jacky	Géographie
HERAULT Jeanny	Physique
HERINO Roland	Physique
JARDON Pierre	Chimie
KERCKHOVE Claude	Géologie
MANDARON Paul	Biologie
MARTINEZ Francis	Mathématiques Appliquées
MOREL Alain	Géographie
NEMOZ Alain	Thermodymamique CNRS - CRTBT
NGUYEN HUY Xuong	Informatique
OUDET Bruno	Mathématiques Appliquées
PAUTOU Guy	Biologie
PECHER Arnaud	Géologie
PELMONT Jean	Biochimie
PELLETIER Guy	Astrophysique
PERRIN Claude	Sciences Nucléaires I.S.N.
PIBOULE Michel	Géologie
RAYNAUD Hervé	Mathématiques Appliquées
REGNARD Jean René	Physique
RICHARD Jean-Marc	Physique
RIEDTMANN Christine	Mathématiques Pures
ROBERT Danielle	Chimie
ROBERT Gilles	Mathématiques Pures
ROBERT Jean-Bernard	Chimie Physique
SARROT-REYNAULD Jean	Géologie
SAYETAT Françoise	Physique
SERVE Denis	Chimie
STOECKEL Frédéric	Physique
SCHOLL Pierre-Claude	Mathématiques Appliquées
SUBRA Robert	Chimie
VALLADE Marcel	Physique
VIDAL Michel	Chimie Organique
VINCENT Gilbert	Physique
VIVIAN Robert	Géographie
VOTTERO Philippe	Chimie

MEMBRES DU CORPS ENSEIGNANT DE L' IUT 1

PROFESSEURS de 1ère Classe

BUISSON Roger	Physique IUT 1
CHEHIKIAN Alain	E.E.A. I.U.T.1
DODU Jacques	Mécanique Appliquée IUT 1
NEGRE Robert	Génie Civil IUT 1
NOUGARET Marcel	Automatique IUT 1
PERARD Jacques	EEA. IUT 1

PROFESSEURS de 2ème classe

BEE Marc	Physique IUT 1
BOUTHINON Michel	EEA. IUT 1
CHAMBON René	Génie Mécanique IUT 1
CHENAVAS Jean	Physique IUT 1

CHILO Jean	Physique IUT 1
CHOUTEAU Gérard	Physique IUT 1
CONTE René	Physique IUT 1
FOSTER Panayotis	Chimie IUT 1
GOSSE Jean-Pierre	EEA.IUT 1
GROS Yves	Physique IUT 1
HAMAR Roger	Chimie IUT 1
KUHN Gérard, (Détaché)	Physique IUT 1
LEVIEL Jean Louis	Physique IUT 1
MAZUER Jean	Physique IUT 1
MICHOULIER Jean	Physique IUT 1
MONLLOR Christian	EEA.IUT 1
PERRAUD Robert	Chimie IUT 1
PIERRE Gérard	Chimie IUT 1
TERRIEZ Jean-Michel	Génie Mécanique IUT 1
TOUZAIN Philippe	Chimie IUT 1
TURGEMAN Sylvain	Génie civil
VINCENDON Marc	Chimie IUT 1
ZIGONE Michel	Physique IUT 1

PROFESSEURS DE PHARMACIE

AGNIUS-DELORD Claudine	Physique	Faculté La Tronche
ALARY Josette	Chimie Analytique	Faculté La Tronche
BERIEL Hélène	Physiologie et Pharmacologie	Faculté La Tronche
CUSSAC Max	Chimie Therapeutique	Faculté La Tronche
DEMENGE Pierre	Pharmacodynamie	Faculté La Tronche
FAVIER Alain	Biochimie	C.H.R.G.
JEANNIN Charles	Pharmacie Galénique	Faculté Meylan
LATURAZE Jean	Biochimie	Faculté La Tronche
LUU DUC Cuong	Chimie Générale	Faculté La Tronche
MARIOTTE Anne-Marie	Pharmacognosie	Faculté La Tronche
MARZIN Daniel	Toxicologie	Faculté Meylan
RENAUDET Jacqueline	Bactériologie	Faculté La Tronche
ROCHAT Jacques	Hygiéne et Hydrologie	Faculté La Tronche
SEIGLE-MURANDI Françoise	Botanique et Cryptogamie	Faculté Meylan
VERAIN Alice	Pharmacie Galénique	Faculté Meylan

MEMBRES DU CORPS ENSEIGNANT DE MEDECINE

PROFESSEURS CLASSE EXEPTIONNELLE ET 1ère CLASSE

AMBLARD Pierre	Dermatologie	C.H.R.G.
AMBROISE-THOMAS Pierre	Parasitologie	C.H.R.G.
BEAUDOING André	Pédiatrie-Puericulture	C.H.R.G.
BEZEZ Henri	Orthopédie-Traumatologie	Hopital SUD
BONNET Jean-Louis	Ophtalmologie	C.H.R.G.
BOUCHET Yves	Anatomie	Faculté La Merci
	Chirurgie Générale et Digestive	C.H.R.G.
BUTEL Jean	Orthopédie-Traumatologie	C.H.R.G.
CHAMBAZ Edmond	Biochimie	C.H.R.G.
CHAMPETIER Jean	Anatomie-Topographique et Appliquée	C.H.R.G.
CHARACHON Robert	O.R.L.	C.H.R.G.
COLOMB Maurice	Immunologie	Hopital sud
COUDERC Pierre	Anatomie-Pathologique	C.H.R.G.
DELORMAS Pierre	Pneumophtisiologie	C.H.R.G.
DENIS Bernard	Cardiologie	C.H.R.G.
GAVEND Michel	Pharmacologie	Faculté La Merci

HOLLARD Daniel	Hématologie	C.H.R.G.
LATREILLE René	Chirurgie Thoracique et Cardiovasculaire	C.H.R.G.
LE NOC Pierre	Bactériologie-Virologie	C.H.R.G.
MALINAS Yves	Gynécologie et Obstétrique	C.H.R.G.
MALLION Jean-Michel	Médecine du Travail	C.H.R.G.
MICOUD Max	Clinique Médicale et Maladies Infectieuses	C.H.R.G.
MOURIQUAND Claude	Histologie	Faculté La Merci
PARAMELLE Bernard	Pneumologie	C.H.R.G.
PERRET Jean	Neurologie	C.H.R.G.
RACHAIL Michel	Hépato-Gastro-Entérologie	C.H.R.G.
DE ROUGEMONT Jacques	Neurochirurgie	C.H.R.G.
SARRAZIN Roger	Clinique Chirurgicale	C.H.R.G.
STIEGLITZ Paul	Anestésiologie	C.H.R.G.
TANCHE Maurice	Physiologie	Faculté La Merci
VIGNAIS Pierre	Biochimie	Faculté La Merci

PROFESSEURS 2ème CLASSE

BACHELOT Yvan	Endocrinologie	C.H.R.G.
BARGE Michel	Neurochirurgie	C.H.R.G.
BENABID Alim Louis	Biophysique	Faculté La Merci
BENSA Jean-Claude	Immunologie	Hopital Sud
BERNARD Pierre	Gynécologie-Obstétrique	C.H.R.G.
BESSARD Germain	Pharmacologie	ABIDJAN
BOLLA Michel	Radiothérapie	C.H.R.G.
BOST Michel	Pédiatrie	C.H.R.G.
BOUCHARLAT Jacques	Psychiatrie Adultes	Hopital Sud
BRAMBILLA Christian	Pneumologie	C.H.R.G.
CHIROSSEL Jean-Paul	Anatomie-Neurochirurgie	C.H.R.G.
COMET Michel	Biophysique	Faculté La Merci
CONTAMIN Charles	Chirurgie Thoracique et Cardiovasculaire	C.H.R.G.
CORDONNIER Daniel	Néphrologie	C.H.R.G.
COULOMB Max	Radiologie	C.H.R.G.
CROUZET Guy	Radiologie	C.H.R.G.
DEBRU Jean-Luc	Médecine Interne et Toxicologie	C.H.R.G.
DEMONGEOT Jacques	Biostatistiques et Informatique Médicale	Faculté La Merci
DUPRE Alain	Chirurgie Générale	C.H.R.G.
DYON Jean-François	Chirurgie Infantile	C.H.R.G.
ETERRADOSSI Jacqueline	Physiologie	Faculté La Merci
FAURE Claude	Anatomie et Organogénèse	C.H.R.G.
FAURE Gilbert	Urologie	C.H.R.G.
FOURNET Jacques	Hépato-Gastro-Entérologie	C.H.R.G.
FRANCO Alain	Médecine Interne	C.H.R.G.
GIRARDET Pierre	Anesthésiologie	C.H.R.G.
GUIDICELLI Henri	Chirurgie Générale et Vasculaire	C.H.R.G.
GUIGNIER Michel	Thérapeutique et Réanimation Médicale	C.H.R.G.
HADJIAN Arthur	Biochimie	Faculté La Merci
HALIMI Serge	Endocrinologie et Maladies Métaboliques	C.H.R.G.
HOSTEIN Jean	Hépato-Gastro-Entérologie	C.H.R.G.
HUGONOT Robert	Médecine Interne	C.H.R.G.
JALBERT Pierre	Histologie-Cytogénétique	C.H.R.G.
JUNIEN-LAVILLAUROY Claude	O.R.L.	C.H.R.G.
KOLODIE Lucien	Hématologie Biologique	C.H.R.G.
LETOUBLON Christian	Chirurgie Générale	C.H.R.G.
MACHECOURT Jacques	Cardiologie et Maladies Vasculaires	C.H.R.G.
MAGNIN Robert	Hygiène	C.H.R.G.
MASSOT Christian	Médecine Interne	C.H.R.G.

MOUILLON Michel	Ophtalmologie	C.H.R.G.
PELLAT Jacques	Neurologie	C.H.R.G.
PHELIP Xavier	Rhumatologie	C.H.R.G.
RACINET Claude	Gynécologie-Obstétrique	Hopital Sud
RAMBAUD Pierre	Pédiatrie	C.H.R.G.
RAPHAEL Bernard	Stomatologie	C.H.R.G.
SCHAERER René	Cancérologie	C.H.R.G.
SEIGNEURIN Jean-Marie	Bactériologie-Virologie	Faculté La Merci
SELE Bernard	Cytogénétique	Faculté La Merci
SOTTO Jean-Jacques	Hématologie	C.H.R.G.
STOEBNER Pierre	Anatomie Pathologique	C.H.R.G.
VROUSOS Constantin	Radiothérapie	C.H.R.G.

INSTITUT NATIONAL POLYTECHNIQUE DE GRENOBLE

Président : Georges LESPINARD **Année 1988**

Professeurs des Universités

BARIBAUD Michel	ENSERG
BARRAUD Alain	ENSIEG
BAUDELET Bernard	ENSPG
BEAUFILS Jean-Pierre	ENSEEG
BLIMAN Samuel	ENSERG
BLOCH Daniel	ENSPG
BOIS Philippe	ENSHMG
BONNETAIN Lucien	ENSEEG
BOUVARD Maurice	ENSHMG
BRISSONNEAU Pierre	ENSIEG
BRUNET Yves	IUFA
CAILLERIE Denis	ENSHMG
CAVAIGNAC Jean-François	ENSPG
CHARTIER Germain	ENSPG
CHENEVIER Pierre	ENSERG
CHERADAME Hervé	UFR PGP
CHOVET Alain	ENSERG
COHEN Joseph	ENSERG
COUMES André	ENSERG
DARVE Félix	ENSHMG
DELLA-DORA Jean	ENSIMAG
DEPORTES Jacques	ENSPG
DOLMAZON Jean-Marc	ENSERG
DURAND Francis	ENSEEG
DURAND Jean-Louis	ENSIEG
FOGGIA Albert	ENSIEG
FONLUPTJean	ENSIMAG
FOULARD Claude	ENSIEG
GANDINI Alessandro	UFR PGP
GAUBERT Claude	ENSPG
GENTIL Pierre	ENSERG
GREVEN Hélène	IUFA
GUERIN Bernard	ENSERG
GUYOT Pierre	ENSEEG
IVANES Marcel	ENSIEG
JAUSSAUD Pierre	ENSIEG
JOUBERT Jean-Claude	ENSPG
JOURDAIN Geneviève	ENSIEG
LACOUME Jean-Louis	ENSIEG
LESIEUR Marcel	ENSHMG
LESPINARD Georges	ENSHMG
LONGEQUEUE Jean-Pierre	ENSPG
LOUCHET François	ENSIEG
MASSE Philippe	ENSIEG
MASSELOT Christian	ENSIEG
MAZARE Guy	ENSIMAG
MOREAU René	ENSHMG
MORET Roger	ENSIEG
MOSSIERE Jacques	ENSIMAG
OBLED Charles	ENSHMG
OZIL Patrick	ENSEEG
PARIAUD Jean-Charles	ENSEEG
PERRET René	ENSIEG
PERRET Robert	ENSIEG
PIAU Jean-Michel	ENSHMG
POUPOT Christian	ENSERG
RAMEAU Jean-Jacques	ENSEEG
RENAUD Maurice	UFR PGP
ROBERT André	UFR PGP
ROBERT François	ENSIMAG
SABONNADIERE Jean-Claude	ENSIEG
SAUCIER Gabrielle	ENSIMAG
SCHLENKER Claire	ENSPG
SCHLENKER Michel	ENSPG
SILVY Jacques	UFR PGP
SIRIEYS Pierre	ENSHMG
SOHM Jean-Claude	ENSEEG
SOLER Jean-Louis	ENSIMAG
SOUQUET Jean-Louis	ENSEEG
TROMPETTE Philippe	ENSHMG
VEILLON Gérard	ENSIMAG
ZADWORNY François	ENSERG

Professeur Université des Sciences Sociales (Grenoble II)

BOLLIET Louis

Personnes ayant obtenu le diplôme d'HABILITATION A DIRIGER DES RECHERCHES

BECKER Monique
BINDER Zdenek
CHASSERY Jean-Marc
CHOLLET Jean-Pierre
COEY John
COLINET Catherine
COMMAULT Christian
CORNUEJOLS Gérard
COULOMB Jean- Louis
DALARD Francis
DANES Florin
DEROO Daniel
DIARD Jean-Paul
DION Jean-Michel
DUGARD Luc
DURAND Madeleine
DURAND Robert
GALERIE Alain
GAUTHIER Jean-Paul
GENTIL Sylviane
GHIBAUDO Gérard
HAMAR Sylvaine
HAMAR Roger
LADET Pierre
LATOMBE Claudine
LE GORREC Bernard
MADAR Roland
MULLER Jean
NGUYEN TRONG Bernadette
PASTUREL Alain
PLA Fernand
ROUGER Jean
TCHUENTE Maurice
VINCENT Henri

Chercheurs du C.N.R.S
Directeurs de recherche 1ère Classe

CARRE René
FRUCHART Robert
HOPFINGER Emile
JORRAND Philippe
LANDAU Ioan
VACHAUD Georges
VERJUS Jean-Pierre

Directeurs de recherche 2ème Classe

ALEMANY Antoine
ALLIBERT Colette
ALLIBERT Michel
ANSARA Ibrahim
ARMAND Michel
BERNARD Claude
BINDER Gilbert
BONNET Roland
BORNARD Guy
CAILLET Marcel
CALMET Jacques
COURTOIS Bernard
DAVID René
DRIOLE Jean
ESCUDIER Pierre
EUSTATHOPOULOS Nicolas
GUELIN Pierre
JOUD Jean-Charles
KLEITZ Michel
KOFMAN Walter
KAMARINOS Georges
LEJEUNE Gérard
LE PROVOST Christian
MADAR Roland
MERMET Jean
MICHEL Jean-Marie
MUNIER Jacques
PIAU Monique
SENATEUR Jean-Pierre
SIFAKIS Joseph
SIMON Jean-Paul
SUERY Michel
TEODOSIU Christian
VAUCLIN Michel
WACK Bernard

Personnalités agrées à titre permanent à diriger des travaux de recherche (décision du conseil scientifique)

E.N.S.E.E.G
CHATILLON Christian
HAMMOU Abdelkader
MARTIN GARIN Régina
SARRAZIN Pierre
SIMON Jean-Paul

E.N.S.E.R.G
BOREL Joseph

E.N.S.I.E.G
DESCHIZEAUX Pierre
GLANGEAUD François
PERARD Jacques
REINISCH Raymond

E.N.S.H.G
ROWE Alain

E.N.S.I.M.A.G
COURTIN Jacques

E.F.P.
CHARUEL Robert

C.E.N.G
CADET Jean
COEURE Philippe
DELHAYE Jean-Marc
DUPUY Michel
JOUVE Hubert
NICOLAU Yvan
NIFENECKER Hervé
PERROUD Paul
PEUZIN Jean-Claude
TAIB Maurice
VINCENDON Marc

Laboratoires extérieurs

C.N.E.T
DEVINE Rodericq
GERBER Roland
MERCKEL Gérard
PAULEAU Yves

Le travail présenté dans cet ouvrage est avant tout le fruit d'une collaboration entre l'université et l'industrie ; je remercie donc le laboratoire T.I.M. 3 de l'I.N.P.G. ainsi que le département S.E.S. de l'entreprise MERLIN GERIN pour m'avoir accueilli dans leurs locaux.

J'adresse également mes très sincères remerciements à :

- Monsieur Jean FONLUPT, professeur à l'I.N.P.G. et directeur du laboratoire A.R.T.E.M.I.S. de Grenoble, de l'intérêt qu'il a témoigné pour ce travail de recherche et de l'honneur qu'il me fait en acceptant la présidence de ce jury ;

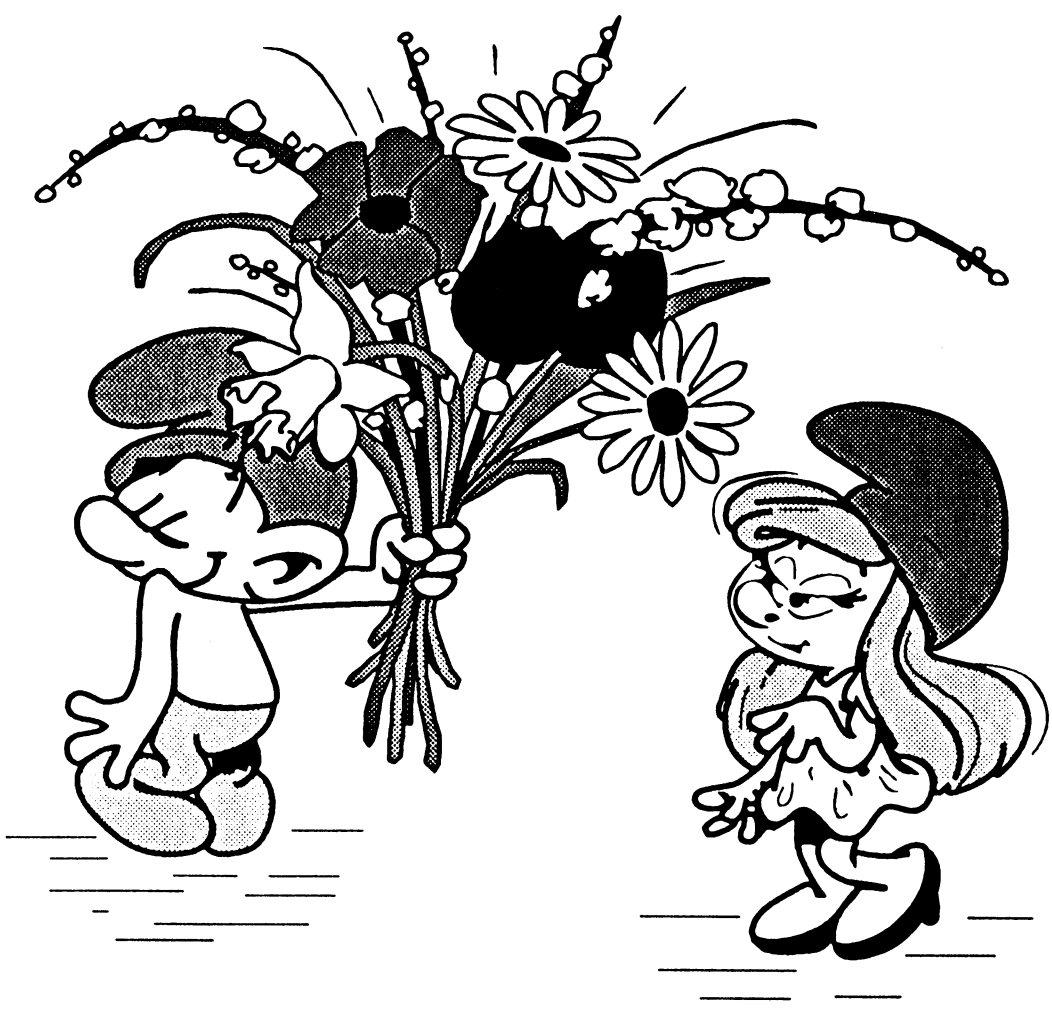
- Monsieur Jean-Louis SOLER, professeur à l'I.N.P.G. et directeur des études de l'E.N.S.I.M.A.G., non seulement pour m'avoir encadré tout au long de cette thèse, mais aussi pour m'avoir fait découvrir l'enseignement ;
- Monsieur Jean-Claude LAPRIE, directeur de recherches au C.N.R.S., qui a bien voulu être rapporteur de ce travail ;
- Monsieur Paul CASPI, chargé de recherches au C.N.R.S., qui a lui aussi accepté d'être rapporteur ; sa compétence ainsi que ses encouragements aux moments cruciaux m'ont permis de terminer ce travail dans d'excellentes conditions morales ;

- Monsieur Eric PILAUD, responsable du groupe logiciel du département S.E.S. à MERLIN GERIN, qui m'a supervisé durant la première moitié de ma thèse ;
- Monsieur Patrick BOUTEILLER, responsable du service technique du département S.E.S. à MERLIN GERIN, pour l'attention constante avec laquelle il a suivi ce travail ;
- Monsieur Jean-Louis BERGERAND, responsable de l'équipe SAGA à MERLIN GERIN, pour avoir toujours trouvé le temps de s'intéresser à mon travail ; ce document ne serait pas sans son engagement personnel et sa sympathie ;

- Monsieur Daniel IGLESIAS et l'équipe de reprographie de l'I.M.A.G. qui ont assuré le tirage de ce document ;
- Monsieur Mac Intosh pour son aide ininterrompue dans la rédaction de cet ouvrage.

Enfin, je remercie tous ceux qui, à MERLIN GERIN comme à l'université, m'ont soutenu dans ce travail.

Table des matières



A ma femme Christine

qui m'aura toujours supporté...

INTRODUCTION

Aujourd'hui, de nombreux processus industriels sont pilotés par des systèmes programmés. Ces processus sont souvent complexes ; parfois, leur défaillance peut entraîner une catastrophe (nucléaire, transport, avionique,...). Les logiciels de contrôle de ces processus deviennent alors souvent eux-mêmes complexes, et leur défaillance peut aussi devenir dangereuse. Pour réduire ce risque, il faut d'abord savoir mesurer la confiance que l'on éprouve vis-à-vis d'un tel logiciel ; en effet, c'est au vu des résultats de mesures que l'on pourra décider si ce logiciel est assez fiable ou s'il nécessite des modifications. Or, contrairement au domaine du matériel pour lequel des techniques bien rodées sont enfin acceptées par la majorité, aucune méthode de validation n'est universellement admise pour quantifier la confiance que l'on éprouve vis-à-vis d'un logiciel.

L'objectif de cette thèse est de proposer une méthode d'évaluation de la complexité des programmes destinés notamment à piloter les équipements de sûreté des centrales nucléaires françaises. Il s'inscrit dans un contexte général de recherches sur la qualité des logiciels, les constructeurs et les utilisateurs de systèmes informatiques ayant en commun le souci de plus en plus grand de quantifier le risque de défaillance d'un système due à une faute logicielle.

Cette étude trouve son origine dans un problème posé par l'entreprise Merlin Gerin[1] qui, par l'intermédiaire de son département Systèmes et Electronique de Sûreté, conçoit et met en place les logiciels de protection des centrales nucléaires civiles françaises. Les logiciels réalisés sont donc hautement critiques, au sens où leur défaillance éventuelle aurait de graves conséquences. Il apparaît donc indispensable de :

- Produire des logiciels fiables :

C'est pourquoi ces logiciels sont développés dans un environnement particulier, grâce à un atelier logiciel baptisé SAGA (Spécification d'Applications et Génération Automatisée) dont la présentation fera l'objet du second chapitre. Les concepteurs n'utilisent pas les langages habituels pour écrire leurs programmes (Pascal, Fortran,...), mais s'aident du guide

[1] Ces recherches ont fait l'objet de la convention C.I.F.R.E. numéro 173/86 entre l'entreprise MERLIN GERIN et le laboratoire IMAG-TIM3.

méthodologique fourni par l'outil SAGA pour les décrire, l'atelier générant automatiquement le code de l'application résultante. Cette démarche doit contribuer à produire des logiciels très structurés, bien documentés, toujours formés de fonctions compatibles entre elles et constituant un ensemble cohérent.

- Mesurer le risque résiduel :

Malgré les performances de cet environnement de développement, il est possible que certaines fautes passent à travers les mailles du filet. On peut penser que ce risque résiduel est d'autant plus élevé que les fonctions manipulées sont complexes.

La criticité des applications gérées nécessite un soin particulier lors de la phase de conception des programmes, concrétisé par l'utilisation d'une forte méthodologie logicielle. Cette méthodologie se caractérise notamment par :
- l'utilisation de langages de haut niveau ;
- des revues d'étude à la fin de chaque phase du cycle de vie ;
- la mise en place d'une équipe de vérification indépendante de l'équipe de conception ;
- des outils informatiques d'aide à la vérification.

La recherche des fautes logicielles s'effectue dès la phase de conception des programmes, plus par l'analyse statique des logiciels que par les tests des programmeurs, donc en amont de ces tests.

Dans ce contexte, l'objectif poursuivi dans le travail que nous présentons ici est de fournir un indicateur de complexité pour les fonctions développées sous l'environnement SAGA, ceci dans deux buts :
- l'optimisation de l'effort de test final par le renforcement de ces tests sur les fonctions les plus complexes (aide à l'obtention d'un logiciel correct) ;
- le repérage précoce des fonctions complexes, susceptibles d'être simplifiées dès la phase de conception (prévention des erreurs). Sur ce dernier point, les études ont montré que la correction d'une faute logicielle s'avère d'autant plus coûteuse qu'elle est décelée tard ; il y a donc là un enjeu économique important.

Cette démarche a déjà été mise en œuvre sur des langages classiques, mais de telles études n'ont encore jamais été réalisées sur des logiciels de type "flots de données" tels que ceux conçus en SAGA. C'est pourquoi l'approche du problème que nous présentons ici est originale.

Deux approches ont été proposées pour quantifier la qualité des logiciels : l'une s'intéresse à leur fiabilité (approche dynamique), et l'autre à leur complexité (approche statique) :

- les modèles de fiabilité consistent à étudier le *comportement* d'un logiciel au cours du temps. Le principe est le suivant : une fois le logiciel prêt à fonctionner, il est mis en service, et on relève les instants successifs d'occurrence de défaillances ; au fur et à mesure de la mise au point, la fiabilité du logiciel a tendance à croître, et la durée écoulée entre deux manifestations d'erreurs consécutives tend à augmenter. On cherche alors, à partir des instants des premières défaillances, à prédire le comportement futur du logiciel, et en particulier la date de la prochaine défaillance. C'est un point de vue a posteriori (il est nécessaire de collecter d'abord des mesures pour appliquer ces théories). Cette technique est généralement associée à l'idée de tests de type "boîte noire", au sens où il n'est pas nécessaire de savoir comment le logiciel est structuré pour étudier sa fiabilité.

- les modèles de complexité : on désigne ainsi toutes les méthodes d'analyse du logiciel lui-même, c'est-à-dire du texte du programme source. Par opposition aux modèles dynamiques, on parle généralement d'approche "boîte claire", ou approche structurelle. Les deux principales études sur lesquelles nous insisterons dans le premier chapitre traitent plus particulièrement de la structure de contrôle des programmes, et des symboles employés dans l'écriture de ces programmes. Ici, c'est le logiciel lui-même qu'on analyse, et non pas son comportement dans le temps. On adopte donc un point de vue a priori, qui sera le notre dans ce travail.

Cette classification ne doit pas être considérée comme exclusive, puisque par exemple des modèles de croissance récents ont réussi à intégrer des notions de complexité [FON-85]. De plus, il arrive souvent que les fabricants de logiciels utilisent ces deux techniques en parallèle pour améliorer la qualité de leurs produits.

Ce document est organisé en quatre chapitres.

Dans le premier chapitre, et pour situer nos travaux par rapport aux études menées dans le domaine, nous rappelons dans un état de l'art sommaire les hypothèses de base de quelques modèles de croissance de fiabilité (Jelinski-Moranda, Musa, Schick-Wolverton, Littlewood,...), ainsi que certaines mesures classiques de la complexité des logiciels, telles que celle de Halstead (étude des symboles employés dans l'écriture du programme) ou de Mac Cabe (étude du graphe de contrôle des applications).

Le second chapitre est consacré à la description de l'atelier logiciel SAGA autour duquel s'articulent ces travaux. Cet environnement de développement, mis au point par la société Merlin Gerin, propose différents outils pour assister et contraindre le programmeur dans sa tâche ; l'outil de conception auquel nous nous intéressons ici, offre à l'utilisateur la possibilité de programmer ses logiciels de façon graphique en s'appuyant sur un langage de type "flots de données" synchrone.

Dans le troisième chapitre, nous proposons une mesure spécifique de la complexité des logiciels développés grâce à SAGA. Cette mesure s'appuie sur l'axiome suivant : "la complexité d'un tout est supérieure à la somme des complexités des parties composant ce tout". Dans un premier paragraphe, nous rappelons la démarche de M.H. Van Emden sur la notion de complexité en l'appliquant à notre cas. Puis dans un second paragraphe, nous formalisons les définitions des termes utilisés tout au long du chapitre. Dans le troisième paragraphe, nous définissons la notion de bloc réalisable, ensuite de quoi nous exposons le mode de calcul choisi pour la métrique, d'abord sur un bloc simple, puis dans le cas général. Nous concluons ce paragraphe par l'application des résultats obtenus sur un exemple. Le quatrième paragraphe est consacré à la définition retenue pour la complexité globale d'une fonction. Dans le cinquième paragraphe, nous étudions un cas réel sur lequel nous montrons que la structuration arborescente des fonctions à l'intérieur d'un logiciel SAGA contribue à diminuer la complexité globale de ce logiciel. Dans le sixième paragraphe, nous proposons une méthode très simple d'approximation de la complexité quand le temps de calcul de cette complexité devient prohibitif : c'est le cas pour les fonctions présentant un grand nombre de variables de même type.

Enfin, nous concluons dans le quatrième chapitre par une étude empirique des résultats de diverses mesures sur des applications SAGA réelles. En effet, cette étude est à notre connaissance la première à porter sur la complexité des logiciels "flots de données" : nous avons pensé qu'il serait intéressant d'en profiter pour prendre des mesures complémentaires à celle définie au chapitre 3. Nous nous bornons dans la première partie de ce chapitre à souligner certains traits caractéristiques de ce genre de programmation. Puis nous présentons une étude de la progression de la complexité d'une fonction SAGA au cours de ses différentes versions, ainsi qu'un modèle de simulation d'arbre de décomposition pour les applications SAGA. Ce chapitre se termine par la présentation de quelques schémas décrivant la progression de la complexité des applications au cours de leur développement.

Les programmes permettant de calculer la métrique présentée dans le troisième chapitre sont rapportés en annexe, ainsi qu'un exemple de fichier de résultats sur l'ensemble des métriques mesurées.

Dans la partie "Références bibliographiques", nous citons l'ensemble des articles auxquels nous nous rapportons dans ce document. Enfin, nous suggérons une bibliographie sur les modèles de croissance de fiabilité, les études générales de complexité, la complexité des logiciels, le flot de données et certains aspects mathématiques et informatiques utilisés ici.

Dans l'ensemble de ce document, nous nous efforcerons d'utiliser la terminologie présentée dans [LAP-89], selon laquelle en particulier :
- une défaillance est la déviation du service délivré par un système par rapport au service spécifié ;
- une erreur est un état du système susceptible de conduire à défaillance ;
- une faute est la cause supposée ou adjugée d'une erreur.

La mesure de complexité présentée dans le troisième chapitre a fait l'objet d'une communication au congrès SAFECOMP 1989 "Safety of control computer systems" à Vienne (Autriche).

Chapitre 1 ETAT DE L'ART

A) Les modèles de croissance

Le premier modèle dynamique de fiabilité du logiciel a été développé aux Etats-Unis en 1972 par Z. Jelinski et P. Moranda ; il permet sous certaines hypothèses d'estimer le nombre initial de fautes contenues dans un programme, au vu du processus d'occurrence des premières défaillances, ainsi que le taux de défaillance du programme et sa fiabilité résiduelle à tout instant [JEL-72].

Les techniques utilisées s'inspirent de celles employées en fiabilité du matériel, puisque l'on attribue à chaque instant un taux de défaillance au logiciel considéré dans son ensemble, tout comme on le ferait pour un composant électronique. L'hypothèse de base de ce modèle est la suivante : à tout instant, le taux de défaillance est proportionnel au nombre d'erreurs résiduelles qu'il contient. Jelinski et Moranda supposent en outre qu'à chaque observation d'une défaillance, une et une seule faute est corrigée.

Pour chaque correction de faute, le modèle prévoit une diminution du taux de défaillance d'une quantité constante, ce qui revient à attribuer à chaque faute le même poids dans le processus d'occurrence des erreurs. Le modèle suppose de plus la perfection du processus de correction, c'est-à-dire qu'à la suite de chaque défaillance, la correction de la faute correspondante n'introduit pas de nouvelles fautes.

En 1975, J. Musa présente le modèle qui porte son nom [MUS-75], inspiré de celui de Jelinski-Moranda. Comme précédemment, il propose une estimation du nombre de fautes présentes dans le programme à l'instant initial ainsi que du temps moyen d'attente jusqu'à la prochaine défaillance à chaque instant.

1978 : G. Schick et R. Wolverton proposent aussi un modèle dérivé de celui de Jelinski-Moranda : ici, le taux de défaillance à tout instant n'est plus seulement proportionnel au nombre de fautes résiduelles dans le programme, mais aussi au temps écoulé depuis la manifestation de la dernière défaillance observée [SCH-78].

En 1981, B. Littlewood et J.L. Verrall introduisent un nouveau type de modèle qui s'attache à quantifier le taux de défaillance à chaque instant [LIT-81]. L'amélioration *probable*

de la fiabilité du logiciel au cours de la phase de mise au point se traduit par une décroissance stochastique du taux de défaillance, considéré comme une variable aléatoire. Selon l'hypothèse fondamentale de ce modèle, le taux de défaillance du programme reste constant entre deux observations consécutives d'erreurs, et apparaît comme la réalisation d'une variable aléatoire de loi Gamma. On ne peut plus alors considérer que le nombre d'erreurs contenues dans le programme est nécessairement fini : c'est la différence majeure de ce modèle avec les précédents.

D'autres modèles ont par la suite été proposés comme celui de J.C. Laprie [LAP-84], ou celui d'O. Gaudoin et J.L. Soler [GAU-88], tous poursuivant le même but, à savoir estimer la fiabilité d'un logiciel à un instant donné en fonction des instants des défaillances successives survenues depuis la date de début de test sur ce logiciel.

Dans [SOL-88] est présentée une modélisation stochastique du processus de défaillances d'un système présentant des fautes de conception (en particulier d'un logiciel) qui évite les hypothèses assez restrictives généralement faites dans les modèles classiques.

B) Les modèles de complexité

A l'inverse des théories présentées sommairement ci-dessus, les modèles de complexité sont indépendants du temps. Dans ce domaine, on ne s'intéresse qu'à la manière dont le logiciel a été écrit, le but recherché étant de fournir des indicateurs pertinents pour mesurer la complexité des logiciels étudiés. Muni de telles mesures, on peut alors s'aider de ces résultats pour classer les différents programmes, et éventuellement revoir ceux qui présentent une complexité trop élevée.

Les modèles de complexité rapportés ici représentent deux tendances distinctes : propriétés de la structure du programme (étude de Mac Cabe) ou propriétés de son texte (étude de Halstead). Nous terminons le chapitre par la description sommaire d'autres modèles existants.

B-1) Etude structurelle - Métrique de Mac Cabe (1976)

Dans son article "A complexity measure" [MAC-76], Mac Cabe décrit une mesure de complexité liée à la structure du graphe de contrôle du programme, et inspirée d'une propriété particulière touchant à l'étude des graphes.

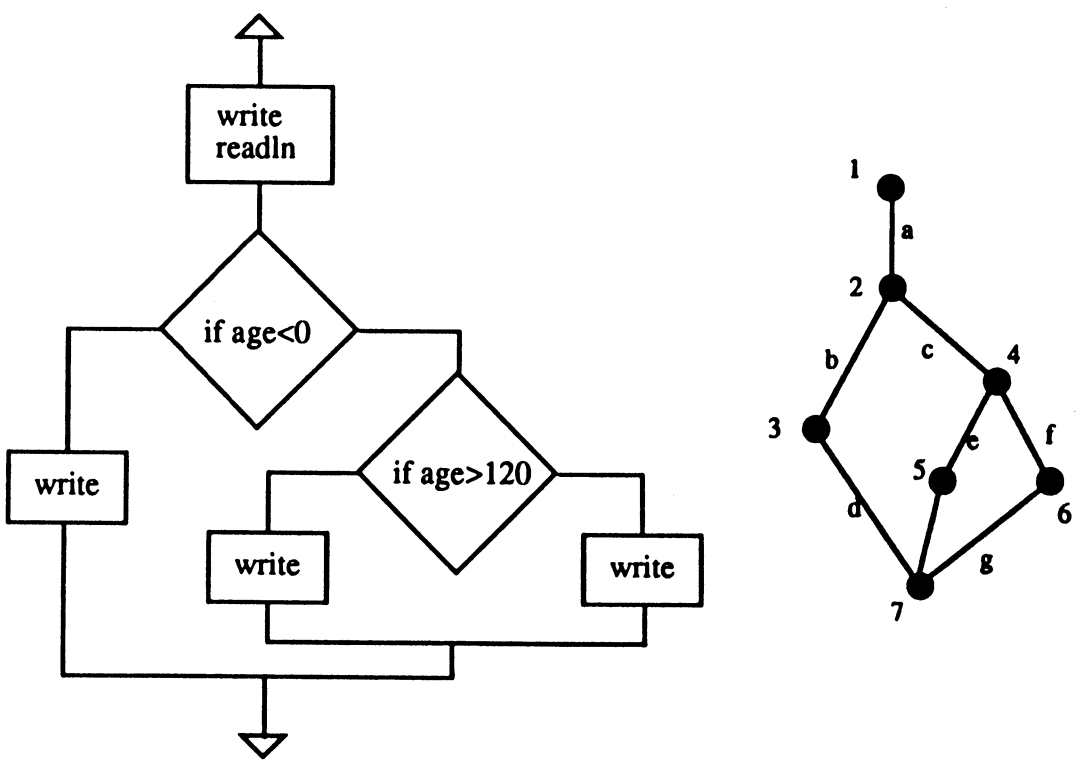
Introduisons d'abord la notion de *graphe de contrôle* d'un programme : on appelle *graphe de contrôle* d'un programme P le graphe orienté G = (V, E) représentant les différents transferts de contrôle à l'intérieur de ce programme. Ce graphe est formé d'un

ensemble de sommets (ou nœuds) V, correspondant aux blocs de code composés d'un ensemble d'instructions exécutées séquentiellement, et d'un ensemble d'arètes E représentant le flot de contrôle entre les divers nœuds de V (instructions de branchement). A tout programme, on peut donc associer son graphe de contrôle, tel que :

- le graphe possède un seul point d'entrée et un seul point de sortie ;
- chaque sommet du graphe représente un bloc d'instructions du programme exécutées de façon séquentielle ;
- chaque arc correspond à un branchement du programme.

```
program VERIF_AGE ;
var age : integer ;
begin
      write('Entrez votre âge : ') ;
      readln(age) ;
      if age<0 then writeln('Age erroné')
      else  if age>120
            then writeln('Vous êtes un phénomène de longévité')
            else writeln('Age enregistré')
end.
```

(Figure 1) Exemple de programme



(Figure 2) Organigramme et graphe de contrôle associés

La mesure définie par Mac Cabe s'appuie sur une des caractéristiques du graphe de contrôle G du programme testé : le nombre cyclomatique v(G), qui exprime le nombre maximal de chemins linéairement indépendants du graphe G. Ce nombre représente aussi le nombre minimal de jeux de tests nécessaires pour activer toutes les branches du programme et également, à une constante près, le nombre de décisions élémentaires contenues dans le programme. L'idée consiste à définir comme complexité du programme ce nombre v(G).

En effet, si on cherche une méthode exhaustive pour certifier un module logiciel, on peut commencer par explorer tous les chemins possibles du graphe de contrôle. Mais l'exemple suivant prouve que le nombre de ces chemins peut être extrêmement élevé ; soit un programme de 50 lignes constitué de 25 structures "IF...THEN...ELSE..." consécutives. Le graphe de contrôle d'un tel module ne comporte pas moins de 2^{25} chemins distincts, soit plus de 33.5 millions de jeux de test.

Il est alors clair qu'on ne peut raisonnablement pas tester le logiciel sur chacun de ces chemins. L'idée de Mac Cabe consiste à qualifier le module grâce au nombre de chemins indépendants du graphe de contrôle.

B-1-1) Notations

G : graphe de contrôle du programme étudié ;
m : nombre d'arcs du graphe G ;
n : nombre de sommets du graphe G ;
p : nombre de composantes connexes du graphe G ;
v(G) : nombre cyclomatique du graphe G.

B-1-2) Résultats

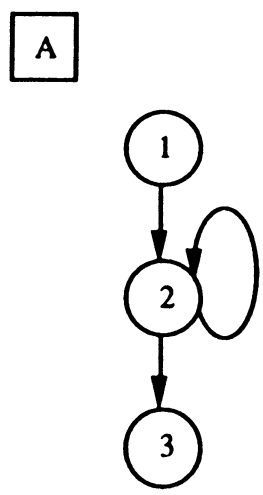
D'après [BER-83], le nombre maximal de cycles élémentaires indépendants de G (ou nombre cyclomatique de G) est défini par :

$$v(G) = m - n + p .$$

Puisque G est orienté, on a le résultat suivant [BER-83] : si G est fortement connexe, alors G admet une base de $v(G) = m - n + 1$ circuits indépendants. Le graphe G étudié ici est trivialement connexe (en effet, toutes les instructions du programme correspondant sont susceptibles d'être exécutées) ; si l'on ajoute alors un arc de retour entre les sommets final et initial (ce qui revient à faire boucler le programme sur lui-même), alors le graphe devient fortement connexe, et la formule s'écrit alors (p=1 et rajout d'une arète)

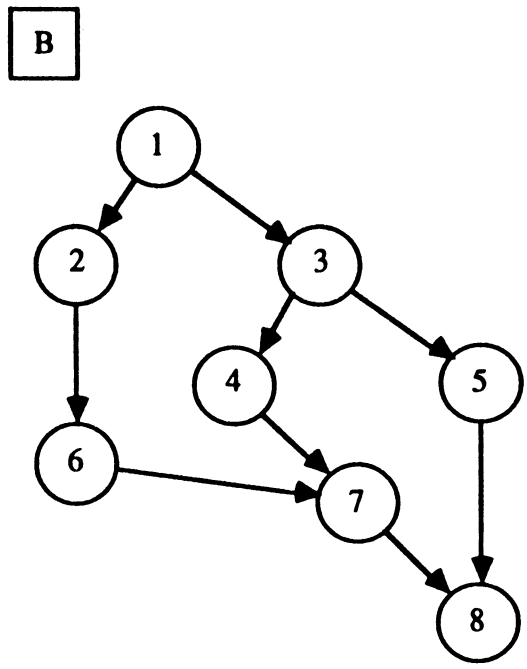
$$v(G) = m - n + 2 .$$

Cette dernière équation donne, selon Mac Cabe, un indicateur de la complexité d'un programme d'après son graphe de contrôle ; ainsi dans l'exemple précédent a-t-on $v(G) = 8 - 7 + 2 = 3$. L'auteur, dans son article [MAC-76], donne de nombreux exemples de graphes de contrôle pour lesquels il établit intuitivement une corrélation entre la métrique développée et l'idée subjective que l'on peut se faire quant à la complexité de tels graphes. Pour mémoire, rappelons quelques exemples :



Dans l'exemple A, on a $n = 3$ sommets et $m = 3$ arètes, donc la complexité est donnée par :

$$v(G) = 3 - 3 + 2 = 2 .$$



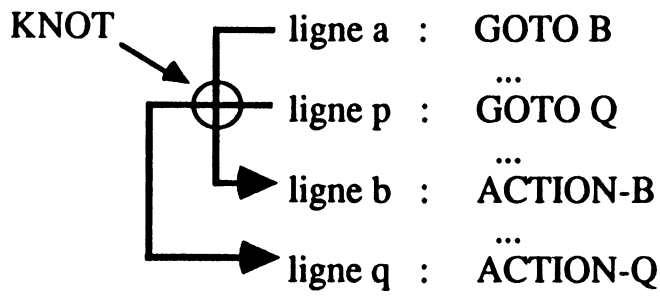
Dans l'exemple B, on a $n = 8$ sommets et $m = 9$ arètes, donc la complexité est donnée par :

$$v(G) = 9 - 8 + 2 = 3 .$$

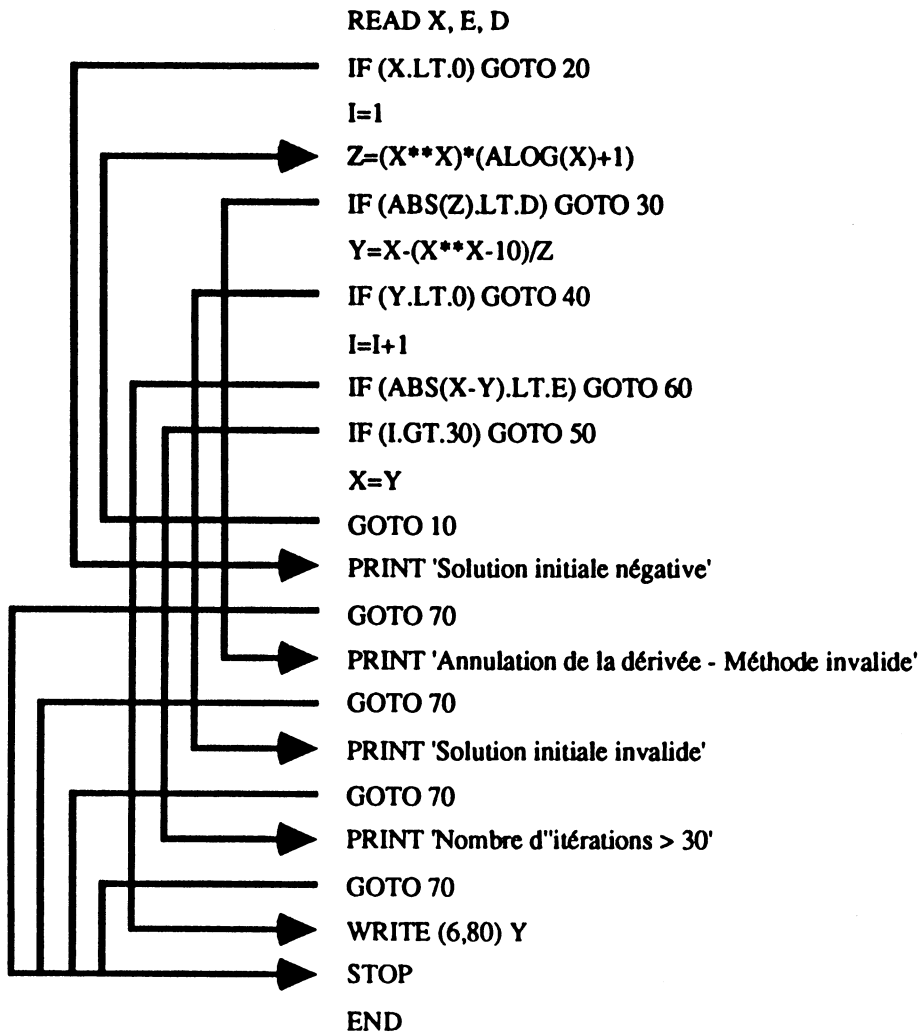
On peut remarquer que :
- si l'on rajoute un sommet sur une des arètes du graphe (ce qui revient à ajouter un bloc de code séquentiel à l'endroit correspondant dans le programme), on ne modifie pas v(G) ;
- le nombre cyclomatique ne dépend que du nombre et de la structure des sommets dits "de décision" (ou de branchement) du graphe.

B-1-3) Modèles dérivés

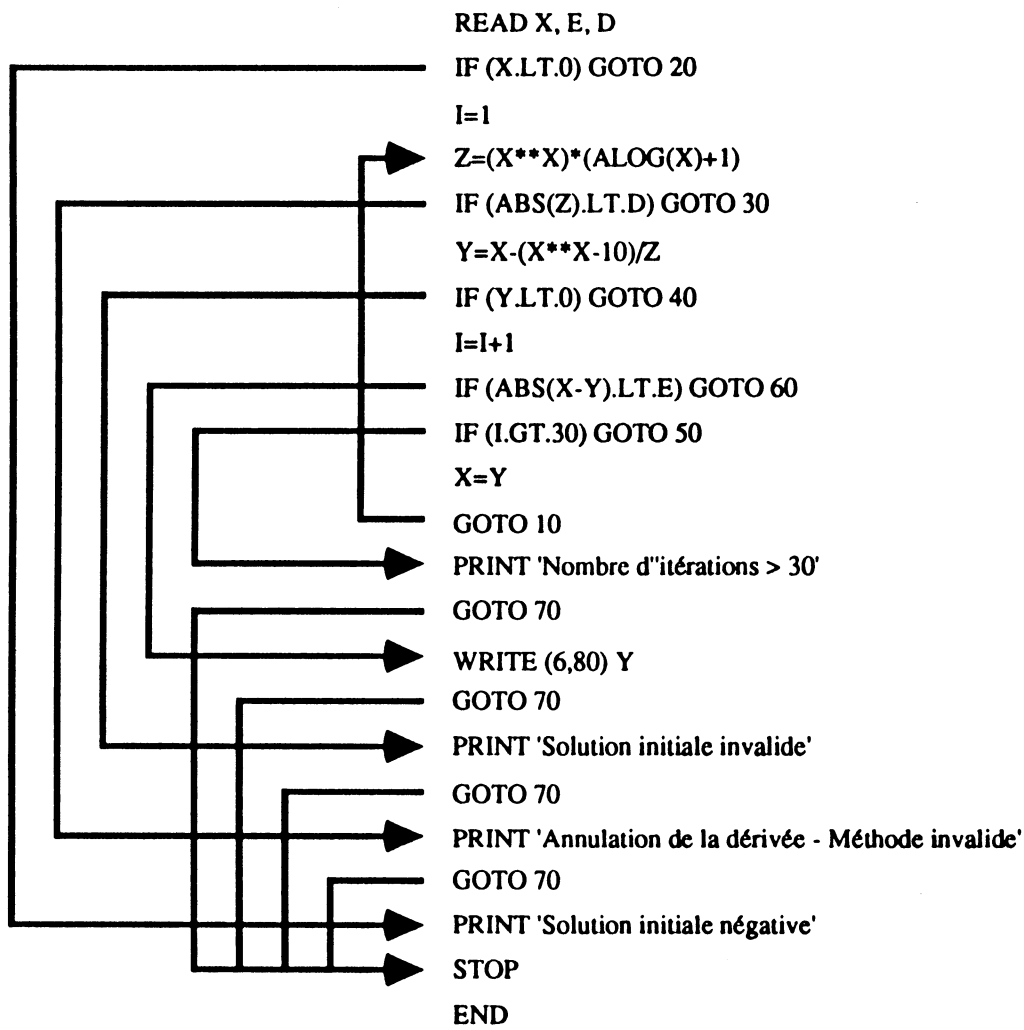
On peut citer ici les travaux de M. Woodward, M. Hennell et D. Hedley sur une mesure de complexité de logiciel baptisée KNOTS [WOO-79]. Le principe de cette métrique consiste à dénombrer les points d'intersection des transferts de contrôle dans un programme (ou "nœuds"), i.e. les configurations du type suivant :



La métrique est définie par le nombre total de nœuds du programme, et caractérise le désordre apparent du graphe de contrôle du programme. Dans l'exemple ci-dessous, on montre comment un programme peut être amélioré du point de vue de cette métrique : on considère un programme permettant de calculer la racine de l'équation "$x^x = 10$" grâce à la méthode de Newton ; on pose $f(x) = x^x - 10$, et on itère $x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}$ jusqu'à ce que la différence entre x_n et x_{n+1} soit inférieure à une certaine limite E ; D désigne la borne inférieure choisie pour la valeur absolue de la dérivée de la fonction f ; X désigne la valeur initiale choisie, puis les résultats successifs des calculs.



Sous cette forme, le programme présente 23 croisements de transfert de contrôle. Si on réorganise les instructions de ce programme, on peut obtenir la nouvelle écriture suivante :



Les deux écritures de ce programme définissent le même algorithme, mais par une simple permutation sur l'ordre des instructions, on a réduit le nombre de croisements de transfert de contrôle à 14. L'idée suggérée est donc de simplifier la compréhension d'un programme en agissant sur la structuration des transferts de contrôle.

Précisons pour terminer que si nous réduisons volontairement le nombre de métriques présentées ici, bien d'autres ont été développées (voir par exemple [HAR-81a], [HAR-81b] ou [LIC-87]).

B-1-4) Conclusions sur ces modèles

L'idée de Mac Cabe consistant à mesurer la complexité de la structure de décision du programme est maintenant universellement utilisée, aussi bien dans l'industrie que dans les universités, mais elle a suscité des critiques : en effet, on peut exhiber deux programmes, l'un facile à interpréter, l'autre plus ardu, qui auront le même nombre cyclomatique associé. Le

résultat de la mesure de complexité de Mac Cabe peut donc être en désaccord avec l'idée qu'un développeur se fait de la complexité de ses programmes.

L'étude du graphe complet permet plus encore de préciser la notion de "style" de programmation, par reconnaissance de structures identiques répétées dans un logiciel par un programmeur [MAC-76].

B-2) Etude textuelle - Métrique de Halstead (1977)

M. H. Halstead a développé dans [HAL-77] une théorie de la complexité des logiciels qui s'appuie sur quelques caractéristiques du texte du programme étudié. Le principe de ce modèle consiste à estimer la complexité d'un logiciel à partir du nombre de symboles (ici, opérandes et opérateurs) utilisés pour l'écriture du programme-source. Ce modèle constitue une application de la théorie de l'information au texte des logiciels à analyser.

Les métriques issues de ce modèle, comme toutes les mesures qui lui ont été associées (dénombrement des lignes du programme, de ses entrées-sorties, de ses procédures...) sont connues sous l'appellation de *métriques textuelles*.

B-2-1) Notations

n_1	:	nombre d'opérateurs distincts du programme ;
n_2	:	nombre d'opérandes distincts du programme ;
N_1	:	nombre total d'opérateurs du programme ;
N_2	:	nombre total d'opérandes du programme ;
$n = n_1 + n_2$	:	nombre de symboles distincts du programme ;
$N = N_1 + N_2$	:	nombre total de symboles du programme.

B-2-2) Résultats

Halstead pose qu'une métrique de complexité des logiciels doit pouvoir s'appliquer à une grande partie des langages utilisés pour mettre en œuvre l'algorithme ; il propose donc de s'attacher simplement aux nombres d'opérandes et d'opérateurs utilisés dans l'écriture du programme, et non pas aux jeux de caractères ou de symboles utilisés.

On peut alors faire la remarque suivante : si l'on code les opérateurs et les opérandes suivant un procédé binaire, alors il est connu que pour repérer un élément quelconque de ce vocabulaire, il nous faudra au moins $Log_2(n)$ informations (ou bits), où n est la taille du vocabulaire. Tout programme comportant N symboles devra donc être codé en moyenne par $N . Log_2(n)$ bits.

Halstead propose alors de définir la mesure suivante pour le volume V du programme :

$$V = N . Log_2(n) = (N_1 + N_2) . Log_2(n_1 + n_2) .$$

Cette dernière équation donne aussi le nombre moyen de discriminations mentales nécessaires à l'écriture d'un programme de N symboles représentant n symboles distincts ; en effet, si on suppose que le programmeur choisit chacun des N symboles qu'il utilise dans l'écriture de son programme parmi n symboles distincts suivant un processus dichotomique, alors il effectue $Log_2(n)$ discriminations pour chacun des symboles.

```
program VERIF_AGE ;
var age : integer ;
begin
      write('Entrez votre âge : ') ;
      readln(age) ;
      if age<0 then writeln('Age erroné')
      else  if age>120
            then writeln('Vous êtes un phénomène de longévité')
            else writeln('Age enregistré')
end.
```

Liste des opérandes		Occurrences	Liste des opérateurs	Occurrences
program		1	VERIF_AGE	1
;		4	age	4
var		1	'Entrez votre âge'	1
:		1	0	1
integer		1	'Age erroné'	1
begin		1	120	1
write		1	'Vous ... longévité'	1
(		4	'Age enregistré'	1
)		4		
readln		1		
if		2		
<		1		
then		2		
writeln		3		
else		2		
>		1		
end		1		
.		1		
18	Totaux	32	8	11

Sur l'exemple ci-dessus, on a un volume $V = (N_1 + N_2) . Log_2(n_1 + n_2)$, soit $V = (32 + 11) . Log_2(18 + 8) = 202.12$.

A partir de ces considérations, il développe une théorie prédisant entre autres le temps nécessaire à l'écriture d'un logiciel, ou le nombre de fautes contenues dans ce logiciel.

B-2-3) Conclusion sur ce modèle

La publication des travaux d'Halstead a engendré plusieurs controverses ; dans [SHE-83] notamment, les auteurs soulignent l'indétermination possible au sujet du classement de certains mots du programme dans la catégorie "opérandes" ou "opérateurs".

Néanmoins V. SHEN, S. CONTE et H. DUNSMORE suggèrent dans [SHE-83] que les relations présentées par Halstead pourraient être des approximations d'équations plus générales reliant les quantités définies en B-1.

B-3) Conclusion sur les modèles de complexité

Comme pour les modèles de croissance, nous nous sommes limités dans notre présentation. Le lecteur trouvera dans [SHA-83] ou [CHA-79] par exemple de plus amples informations sur les modèles de complexité.

Les métriques de Halstead, comme celles de Mac Cabe, ont une très large audience chez les concepteurs de logiciels. Ceux-ci considèrent d'ailleurs que ces méthodes et les modèles de croissance sont complémentaires, et utilisent les unes et les autres en parallèle. Pour les gros projets (transports, aéronautique, nucléaire,...), les procédures de collecte de ces données sont très formalisées, et les conclusions tirées de leurs résultats sont exploitées pour la gestion des projets ultérieurs.

C) Notre choix et ses raisons

Nous avons choisi de nous intéresser aux modèles de complexité, et ceci pour trois raisons principales :

- d'abord, on peut noter que les modèles de croissance reposent sur des résultats d'études statistiques. Pour que ces études soient valables, il est nécessaire de collecter un nombre important de données, ce qui impose de tester le logiciel pendant une durée assez longue avant de pouvoir fournir des informations concernant sa qualité. Mais dans un contexte nucléaire, du fait de la criticité des logiciels produits et donc du soin particulier apporté lors de leur conception, les données de défaillance sont très peu nombreuses, et toute étude statistique sur les résultats de telles mesures serait donc discutable, malgré l'expérience acquise sur des produits logiciels similaires.
- ensuite, il nous semble essentiel dans les domaines de haute sûreté comme le nucléaire, la défense ou les transports, que l'on puisse définir le plus tôt possible des critères de qualité applicables à tous les niveaux de la conception ;
- enfin, la validité du modèle dynamique choisi dépend de l'aptitude de ce modèle à décrire le comportement du logiciel, mais dépend probablement aussi des méthodes de conception et de développement employées. De même le déroulement des tests diffère suivant le niveau de criticité des logiciels à produire. La méthodologie de spécification et de conception influe sans aucun doute sur la fiabilité des logiciels produits, et doit à notre sens être prise en compte dans le modèle choisi.

Ces remarques nous ont donc amenés à nous intéresser plus particulièrement aux modèles basés sur l'étude **structurelle** des logiciels. Ces métriques sont en effet ressenties

comme des mesures "de bon sens" : il y a manifestement corrélation entre notre intuition de ce que doit être la complexité d'un programme, et les résultats de ces métriques. Les programmeurs professionnels reconnaissent, en outre, qu'intuitivement il existe une corrélation entre la complexité subjective d'un programme et la propension qu'il a à contenir des fautes ; MM. ALBIN et FERREOL ont d'ailleurs confirmé scientifiquement cette idée ; on trouve en effet dans [ALB-82] le tableau suivant, représentant les valeurs moyennes prises par les métriques définies ci-dessus pour les procédures avec fautes et celles sans fautes :

	N	n	V	v_N
Toutes procédures	131.2	33.5	771.9	0.067
Procédures sans erreurs	72.3	23.5	374.1	0.069
Procédures avec erreurs	238.7	51.6	1447.2	0.061

Dans le tableau ci-dessus, N désigne la taille des programmes, n leur vocabulaire (cf. [HAL-77]), V leur volume et v_N leur nombre cyclomatique normalisé (par rapport à la taille du programme). Au vu de ces résultats, il semble possible de prévoir quelles fonctions vont receler des erreurs en fonction de leurs mesures de complexité, puisque trois sur quatre sont bien discriminantes. Mais il faut aussi tenir compte dans le modèle de l'environnement de développement : l'étude présentée dans [ALB-82] portait sur la phase de développement du compilateur d'un langage temps réel ; les résultats ne sont pas immédiatement transposables sur un projet de pilotage d'avion militaire ou sur celui d'une centrale nucléaire.

Néanmoins, l'hypothèse qui sous-tend notre étude reste que la probabilité de commettre une erreur de conception dans la définition d'une fonction logicielle augmente dans le même sens que la complexité de cette fonction.

Il nous faut maintenant choisir une mesure de complexité satisfaisante pour les logiciels développés avec l'outil SAGA. Dans notre cas, il ne serait pas judicieux de choisir la métrique de Mac Cabe parce que le langage utilisé est de type "flots de données" (dans un tel langage, toutes les branches du logiciel sont actives à chaque instant, et il n'y a pas de notion de transfert de contrôle).

Enfin, les métriques de complexité sont appliquées sur des modules de programme : il existe peu de modèles proposant de valider un programme global, formé par l'assemblage de modules interagissant entre eux.

Il est difficile de répondre à ces questions. Cependant, il nous semble que la diversité des langages pour lesquels ces métriques sont proposées contribue encore à accentuer cette difficulté ; notre projet consiste à proposer une nouvelle métrique adaptée exclusivement à un

type de programmation précis, de façon à pouvoir davantage utiliser les spécificités du langage utilisé.

Dans le chapitre suivant, on présente l'outil SAGA de conception de logiciels, sur lequel se base notre travail.

Chapitre 2 DESCRIPTION DE L'ATELIER SAGA

SAGA [BGD-87] est un atelier de conception de logiciels. Il a été conçu afin d'assurer la meilleure qualité possible pour des logiciels complexes traitant de problèmes temps réel. Les logiciels visés sont les systèmes à haut degré de sûreté, principalement dans le domaine du nucléaire civil et militaire.

Le formalisme de l'outil SAGA est adapté à l'automatisme temps réel pour les applications cycliques. Le langage intégré à l'outil SAGA, à vocation spécifiquement temps réel, a été créé pour permettre une interprétation temporelle très simple des programmes produits. Le temps réel est en effet un domaine particulier de l'informatique qui comprend l'ensemble des applications nécessitant une interaction continue entre les systèmes de contrôle et l'environnement physique. L'automatisme, le contrôle de processus industriels, le traitement du signal sont des exemples de telles applications.

Les systèmes temps réel sont différents des autres systèmes, car ils possèdent des caractéristiques inhérentes aux propriétés des processus physiques avec lesquels ils interagissent : possibilité d'interruption volontairement réduite voire inexistante, évolution continuelle, irréversibilité.

La programmation des systèmes temps réel est donc un problème délicat : en plus des spécifications comportementales en fonction desquelles ils sont écrits, les programmes doivent respecter des contraintes temporelles indispensables au bon fonctionnement du système global.

Il y a encore quelques années, cette programmation se faisait en langage assembleur. Cette solution présentait l'avantage indiscutable de pouvoir gérer finement le temps. Mais elle avait aussi l'inconvénient majeur d'offrir un très faible niveau d'abstraction et d'interprétation au concepteur, ce qui rendait la programmation des systèmes longue, leur mise au point difficile, et leur évolution complexe.

Face à ce problème, on a développé des langages de haut niveau possédant les qualités requises pour la description du comportement temporel des programmes : l'outil SAGA en fait partie.

SAGA comporte cinq outils :

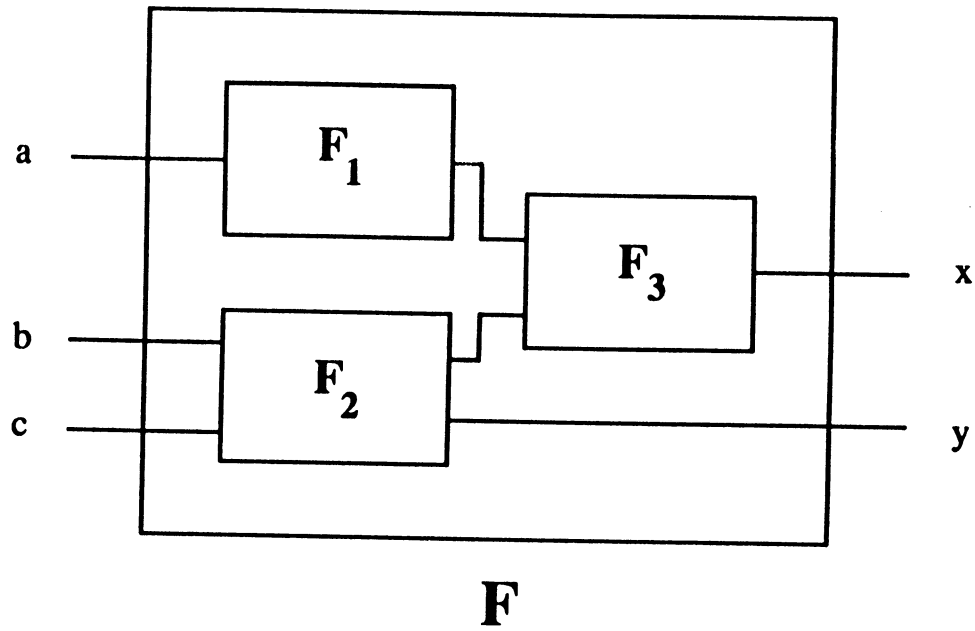
- Conception des fonctions de haut niveau, que nous aborderons en détail ci-dessous ;
- Programmation des fonctions de base, qui permet de coder une fonction dans le langage hôte (ici le langage C) en respectant l'interface définie dans l'outil de Conception ;
- Génération automatique du code correspondant à l'ensemble de l'application, outil qui peut être considéré comme un compilateur, au sens où il permet de générer le code séquentiel correspondant à la description "flots de données" définie grâce à l'outil de Conception ;
- Documentation automatisée, qui permet de mettre en forme l'ensemble des informations fournies par l'utilisateur lors de la définition des fonctions et des données ;
- Administration, qui permet de gérer l'ensemble des applications développées avec l'atelier, de créer de nouvelles applications, et de gérer les droits d'accès des utilisateurs sur ces différentes applications.

Nous nous intéresserons ici essentiellement à l'outil de Conception : cet outil présente trois caractéristiques principales :

- un aspect interactif graphique ;
- une démarche de conception descendante ;
- un langage sous-jacent de type "flots de données" synchrone. Ce type de langage est aussi présenté dans la littérature sous la dénomination de "réactif" [BRY-89].

Une interface graphique : chaque fonction est représentée par une boîte comportant des entrées et des sorties, et composée de sous-fonctions dont on connecte les entrées et les sorties grâce à une souris, de la même façon qu'on connecte des composants électroniques sur une carte en CAO.

On obtient alors pour chaque fonction une représentation sous forme de réseaux des sous-fonctions la composant (figure 3), connectées entre elles par des fils représentant les données circulant entre ces sous-fonctions : on parle aussi de "vue SAGA" ou de "boîte SAGA".



(Figure 3) Exemple de fonction SAGA

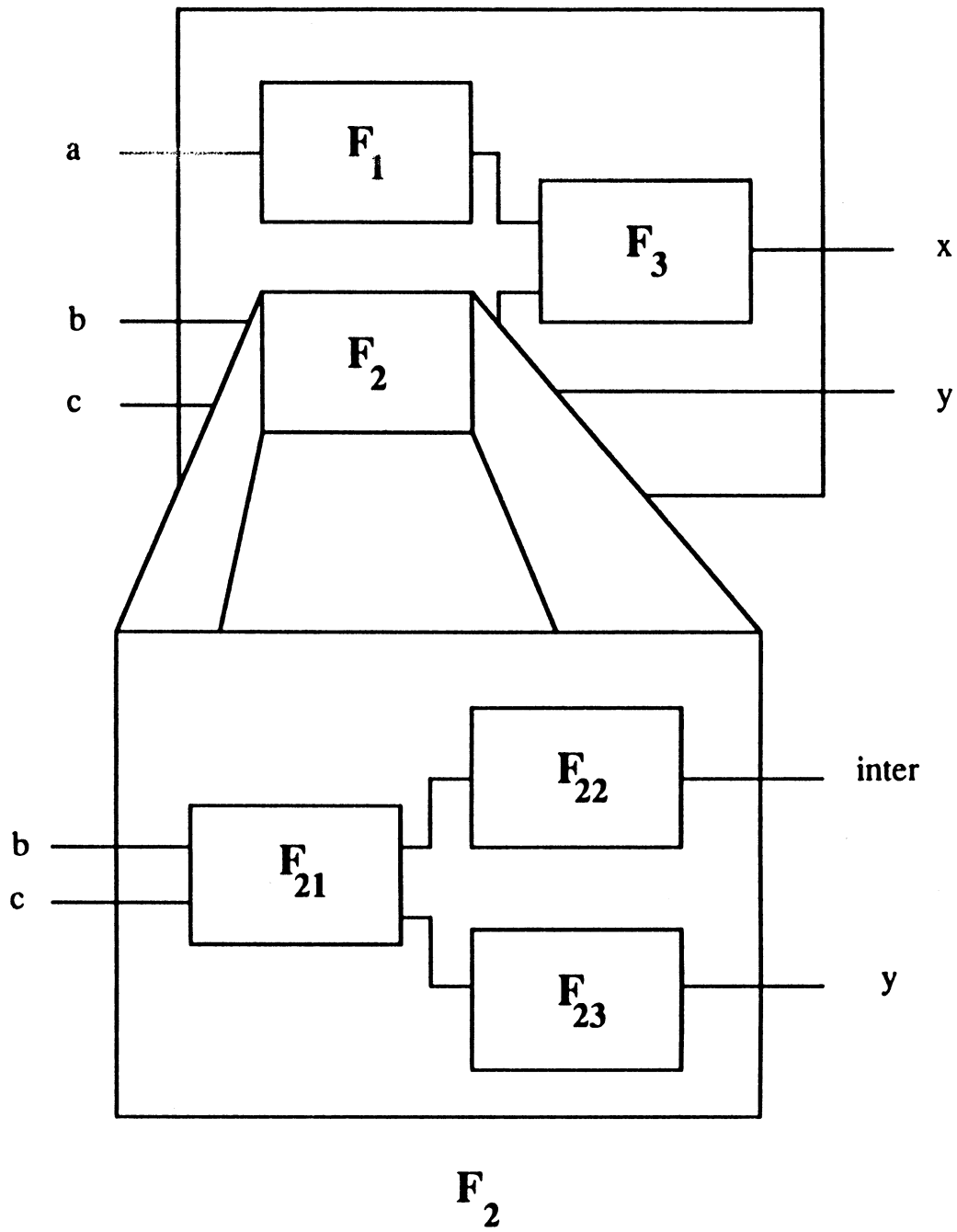
Une démarche de conception descendante : chaque fonction en cours d'affinement est décomposée en un ensemble de sous-fonctions qui concourent à la définition de cette fonction suivant un schéma de type "flots de données", les sous-fonctions utilisées étant elles-mêmes décomposées de façon itérative suivant le même principe. Cette décomposition s'arrête dès que l'une des trois conditions suivantes est vérifiée :

a) la fonction n'est composée que d'opérateurs du langage SAGA (addition, division,...) ;

b) la fonction est programmée directement dans le langage hôte ;

c) la fonction existe déjà dans une bibliothèque sous la forme (a) ou (b) : elle peut alors être réutilisée.

On peut ainsi affiner la description d'une des sous-fonctions composant la fonction principale F de la figure 3, par exemple F_2, de la manière suivante :



(Figure 4) Décomposition d'une fonction SAGA

Un langage de type "flots de données" : nous ne donnons ici qu'un aperçu des caractéristiques de ce type de langage ; le lecteur désirant de plus amples précisions pourra consulter [BGD-86].

Il existe deux approches principales de programmation.

- l'approche classique ou impérative, dans laquelle l'utilisateur décrit la suite des actions à réaliser par la machine pour calculer les résultats à partir des données : c'est le cas des langages usuels comme FORTRAN, PASCAL ou C.

- l'approche déclarative, dans laquelle l'utilisateur manipule exclusivement les données à traiter, et les transformations à appliquer à ces données, le mode et le séquencement des calculs incombant au compilateur.

Toutes les variables apparaissant dans la description d'une fonction sont ainsi définies par une équation unique reliant cette variable aux autres variables présentes dans la vue. Cette équation présente un aspect temporel essentiel, au sens où la définition des variables par leur équation est valable indéfiniment. Par exemple, dans la figure 3, on a :

$$(x, y) = F(a, b, c) .$$

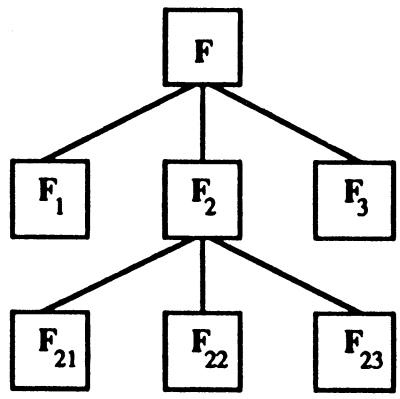
A tout instant t , la valeur des variables x et y dépend des valeurs prises par les variables a, b et c depuis le début du programme, et est déterminée par l'équation

$$(x(t) , y(t)) = F(a([0,t]) , b([0,t]) , c([0,t])).$$

L'aspect "flots de données" du langage utilisé est complété par un typage fort ; c'est l'un des éléments essentiels du contrôle effectué par l'outil SAGA. Toute donnée SAGA doit obligatoirement être déclarée avant de pouvoir être utilisée ; on doit alors notamment préciser son type. La vérification de type peut donc avoir lieu dès la tentative de connexion d'une donnée à l'entrée ou à la sortie d'une fonction.

Ainsi, seules des variables représentant le même type peuvent être connectées entre elles ; par exemple, sur la figure 4, la sortie de la fonction F_{22} est nécessairement du même type que la seconde entrée de la fonction F_3. Cet aspect sera à la base de notre étude.

Le résultat de la définition itérative d'une fonction consiste en l'arbre de décomposition de cette fonction. Dans notre exemple, on obtient pour la fonction F le graphe présenté dans la figure 5.



(Figure 5) Graphe de décomposition d'une fonction SAGA

Comme présenté précédemment, les feuilles de cet arbre sont des fonctions non décomposables, soit parce qu'elles résultent simplement d'une combinaison d'opérateurs du langage SAGA, soit parce qu'elles sont programmées directement dans le langage hôte.

Dans la suite de ce document, nous appellerons application SAGA toute fonction qui n'apparaît dans aucune décomposition d'autres fonctions SAGA. C'est donc une fonction qui se trouve à la racine d'un arbre de décomposition, soit l'équivalent d'un programme au sens classique du terme. Une application SAGA est donc formée de plusieurs fonctions, chacune de ces fonctions étant elle-même éventuellement composée d'autres fonctions.

Notre objectif consiste à évaluer la complexité de chacune des fonctions de l'arbre. Pour une fonction non décomposable, nous pouvons conclure rapidement :
- soit elle consiste en un simple opérateur SAGA, réputé juste : dans ce cas, on décide qu'elle ne contribue pas à la complexité de la décomposition ;
- soit elle est assez simple pour être programmée directement dans le langage hôte, auquel cas on peut appliquer les métriques classiques [MAC-76] [HAL-77].

Nous sommes donc ramenés à l'évaluation de la complexité d'une fonction décomposable. Pour cela, nous allons nous intéresser à la complexité des relations existant entre les sous-fonctions d'une même fonction.

Chapitre 3 COMPLEXITE DES LOGICIELS SAGA

Ce chapitre constitue la majeure partie du document. Dans son introduction, nous présentons l'idée qui nous a guidés dans la définition de la complexité, puis nous appliquons cette idée aux logiciels produits grâce à l'atelier SAGA. Après une étape de formalisation, nous exposons le calcul de la complexité pour chaque fonction, puis pour une application SAGA. Enfin, nous présentons un exemple tiré d'un projet nucléaire, et nous terminons par un calcul très simple d'approximation de la complexité d'une fonction dans des cas limites.

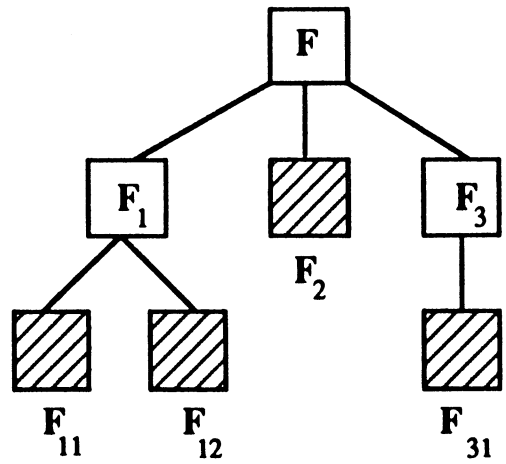
Dans [VAN-75, p.7], on trouve la définition suivante :

"La complexité d'un tout est la façon dont celui-ci diffère de la composition de ses parties".

Van Emden postule en outre que cette différence va toujours dans le sens de la complexité croissante lors de la composition de plusieurs parties pour former un tout ; il propose encore un modèle additif, selon lequel la somme des complexités des sous-systèmes est inférieure ou égale à celle du système entier, la différence entre ces deux quantités mesurant la complexité des relations entre les sous-systèmes.

Si on exhibe une mesure acceptable pour ces interactions, alors on peut définir la complexité d'un tout comme la somme des complexités de ses parties considérées chacune séparément, plus la complexité des interactions entre ces parties.

Ainsi dans la figure 6, la fonction F est composée des 3 sous-fonctions F_1, F_2 et F_3, la fonction F_1 étant elle-même composée des 2 sous-fonctions F_{11} et F_{12}, et la fonction F_3 de F_{31} (on dira qu'une fonction F est composée des sous-fonctions F_1, F_2 et F_3 si la fonction F utilise les résultats de ces sous-fonctions : ainsi la fonction F_3 peut être composée de l'unique sous-fonction F_{31} dont elle utilise les résultats pour élaborer son propre calcul).



(Figure 6) Exemple de décomposition de fonction

Soit C(F) la complexité de la fonction F et R la complexité des interactions entre fonctions ; nous postulons alors en suivant [VAN-75, p.8] :

$$C(F) = R(F ; F_1, F_2, F_3) + C(F_1) + C(F_2) + C(F_3),$$

où $R(F ; F_1, F_2, F_3)$ est la complexité d'assemblage des sous-fonctions F_1, F_2 et F_3 pour former la fonction F.

Si nous appliquons la même formule à F_1 et F_3, nous obtenons :

$$C(F) = R(F ; F_1, F_2, F_3) + [R(F_1 ; F_{11}, F_{12}) + C(F_{11}) + C(F_{12})] + C(F_2) + [R(F_3 ; F_{31}) + C(F_{31})].$$

D'une manière générale, on a $C(F) = \sum_{\sigma \in D(F) \setminus B(F)} R(\sigma ; S(\sigma)) + \sum_{\sigma \in B(F)} C(\sigma)$, où $D(F)$ désigne l'ensemble de toutes les sous-fonctions présentes dans l'arbre de décomposition de la fonction F, $B(F)$ l'ensemble des sous-fonctions non décomposables de cet arbre, et $S(\sigma)$ l'ensemble des sous-fonctions composant directement la fonction σ.

Dans la figure 6, nous obtenons par exemple

$$B(F) = \{ F_{11}, F_{12}, F_2, F_{31} \},$$

$$D(F) = \{ F_1, F_{11}, F_{12}, F_2, F_3, F_{31} \}, \text{ et}$$

$$S(F) = \{ F_1, F_2, F_3 \}.$$

D'après la formule ci-dessus, le calcul de la complexité d'une fonction quelconque se ramène à celui des complexités des fonctions terminales de l'arbre de décomposition (donc non décomposables) et des complexités d'assemblage des différentes sous-fonctions entre elles. C'est ce dernier point auquel nous nous intéresserons dans ce document, la complexité des fonctions terminales constituant un problème à part entière plus fortement dépendant du domaine étudié.

Dans notre cas, tout programme SAGA peut être représenté par un arbre dans lequel chaque sommet représente une fonction utilisée dans la définition de l'application, et composée des sous-fonctions représentées par les sommets successeurs dans l'arbre.

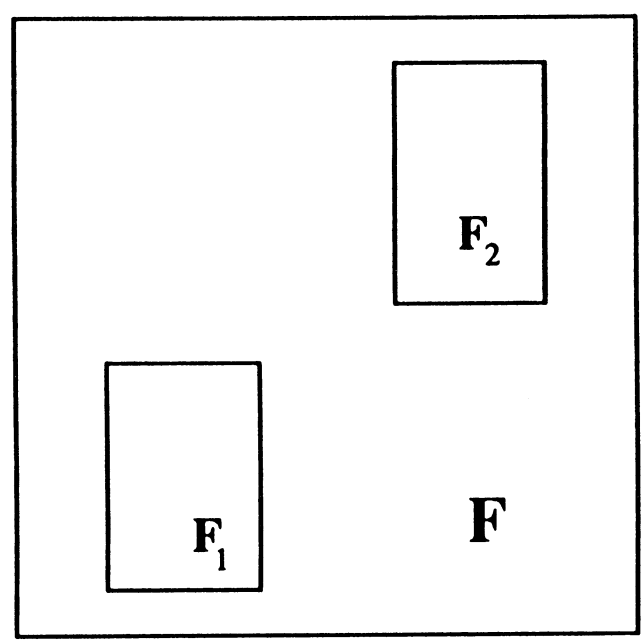
Nous nous inspirons ici des idées de Van Emden, le problème que l'on se pose étant la détermination d'une mesure de la complexité des relations entre fonctions SAGA, ou complexité d'assemblage. La mesure proposée devra également être valable pour des fonctions composées uniquement d'opérateurs SAGA. En revanche, nous n'étudierons pas le cas des fonctions de base développées directement en langage C, tout simplement parce que les métriques existantes (cf. chapitre précédent) nous semblent déjà assez bien adaptées pour pouvoir rendre compte de la complexité de ces fonctions.

A) Application de la démarche de Van Emden à SAGA

Intuitivement, la profondeur totale de décomposition, la complexité visuelle des graphes de connexions (croisements des flots des données, disposition des sous-fonctions,...) et d'autres caractéristiques de la fonction influent sur sa complexité.

Il est clair, au vu des remarques du chapitre 2, que les types des variables contribuent aussi à la complexité de la fonction, au sens où plus les types des variables à connecter dans une vue diffèrent, moins le risque de se tromper dans les connexions est important, puisque SAGA interdit les connexions entre deux variables de types différents. Nous allons donc nous intéresser plus particulièrement à l'aspect "complexité des connexions".

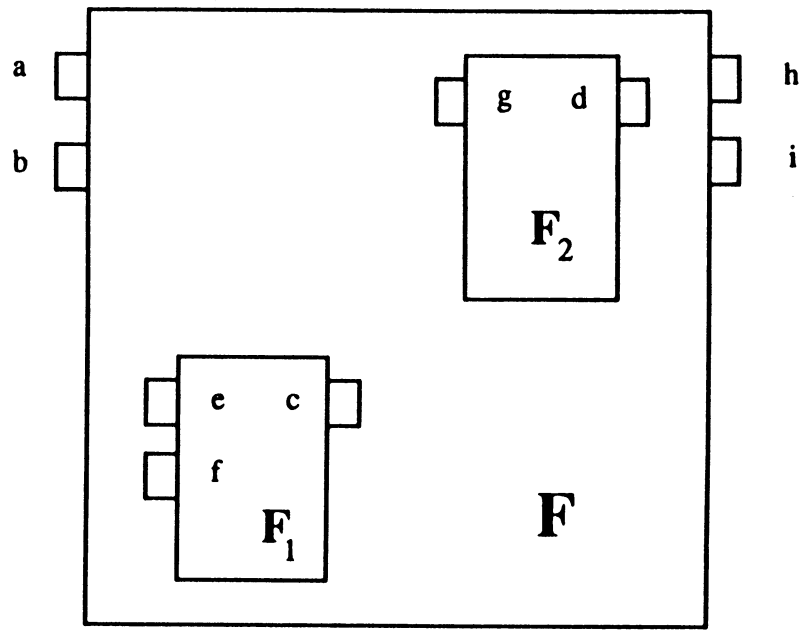
Prenons l'exemple suivant :



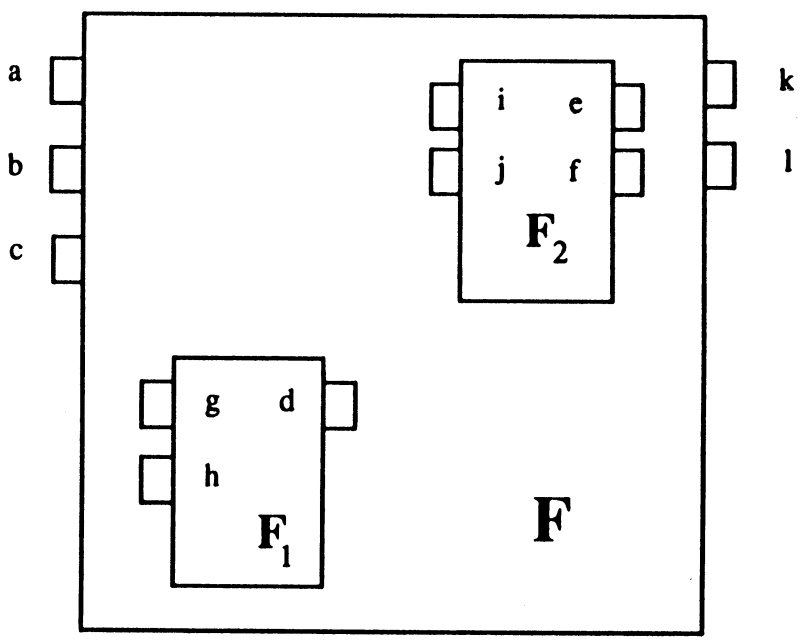
(Figure 7) Description initiale de la fonction F

Dans la figure 7, nous avons fait figurer une fonction SAGA, soit F, ainsi que les deux sous-fonctions la composant, F_1 et F_2. Pour terminer la définition de la fonction F, il reste alors à déterminer les connexions entre toutes les variables présentes dans la vue.

On conçoit facilement que le nombre de connexions possibles entre ces différentes fonctions soit un élément déterminant dans le calcul de la complexité de F. En effet, considérons les deux interfaces suivantes de la fonction F :

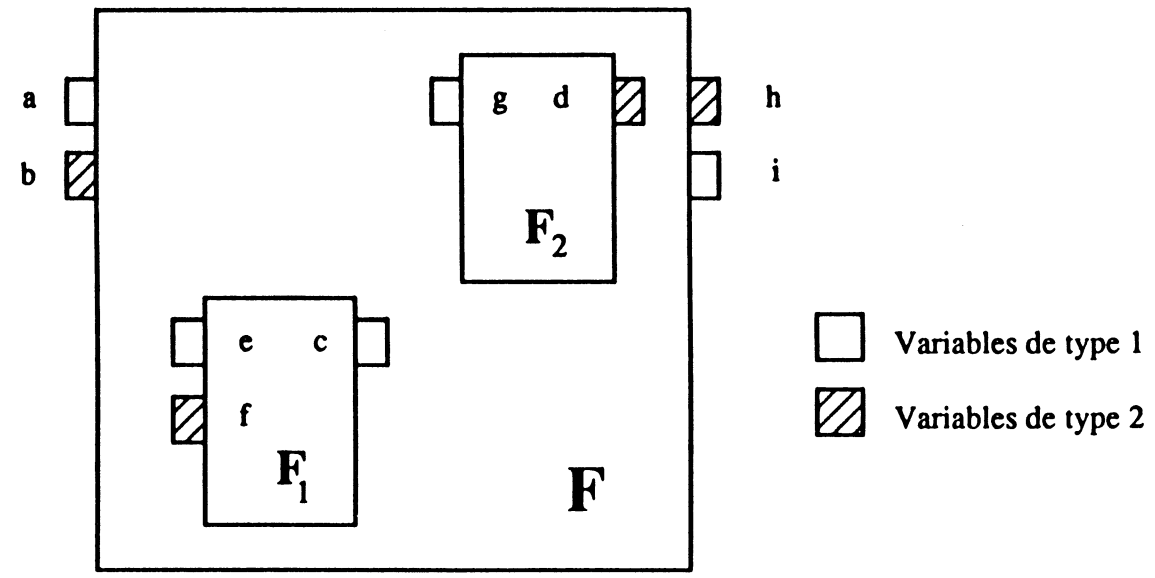


(Figure 8) Description initiale de F - Interface 1

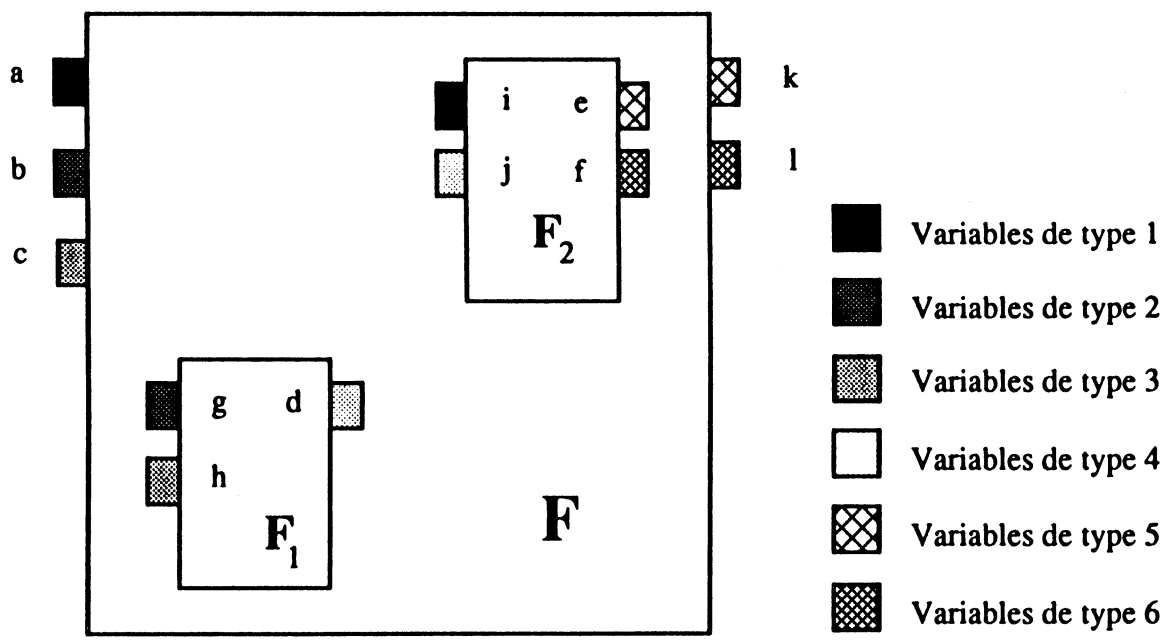


(Figure 9) Description initiale de F - Interface 2

Terminer la définition de F dans la figure 9 semble plus complexe que le faire dans la figure 8, parce que la figure 9 comporte un plus grand nombre de variables. Mais cette remarque ne prend pas en compte les possibilités de contrôle de l'outil SAGA. En effet, considérons maintenant les deux mêmes configurations, typées :

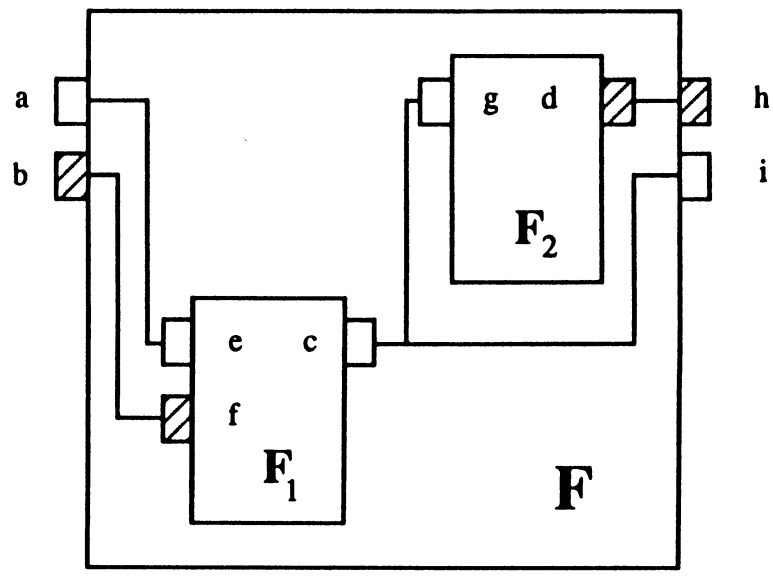


(Figure 10) Description typée de F - Configuration 1

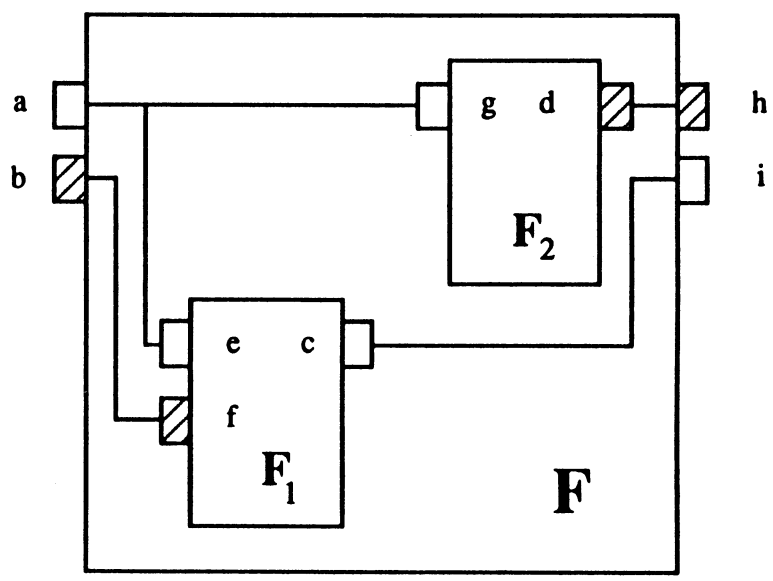


(Figure 11) Description typée de F - Configuration 2

Sur les deux figures ci-dessus, les motifs représentent les différents types de variables utilisées. On constate que dans le premier cas, il existe plusieurs manières de relier entre elles les variables (on dit alors que l'on construit un graphe de connexions pour la fonction F), comme par exemple les deux graphes de connexions suivants (nous verrons dans le paragraphe suivant que ce sont néanmoins les deux seuls autorisés) :

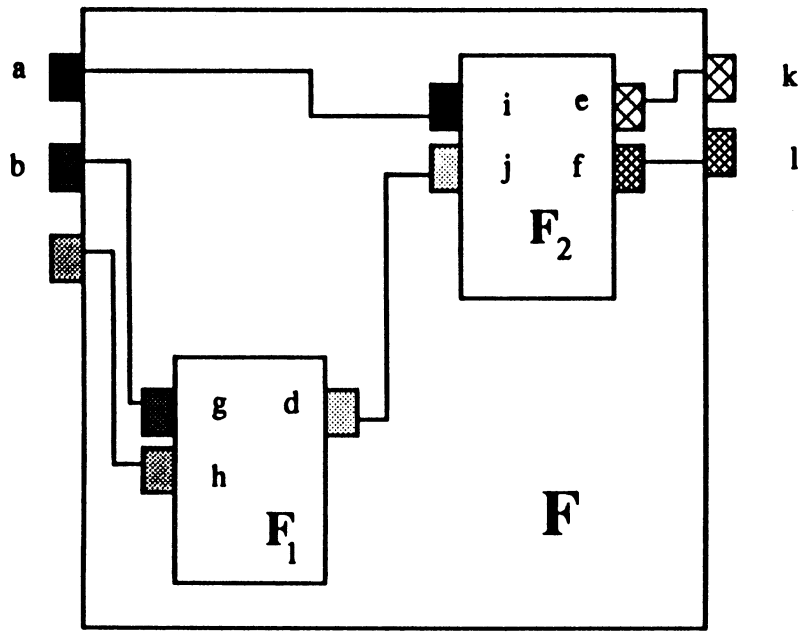


(Figure 12) Configuration 1 - Graphe de connexions 1



(Figure 13) Configuration 1 - Graphe de connexions 2

En revanche, pour la seconde configuration (figure 11), il n'existe qu'un seul graphe de connexions possible pour la fonction F, présenté ci-dessous :



(Figure 14) Configuration 2 - Graphe de connexions

Si l'on se réfère à l'hypothèse fondamentale du chapitre 1 (C) selon laquelle la probabilité de se tromper en définissant une fonction augmente avec la complexité de cette fonction, nous constatons sur cet exemple qu'il est important de prendre en compte dans le calcul de la complexité non seulement le nombre total de variables de la fonction, mais encore plus significativement le nombre total de graphes de connexions permis par l'outil SAGA (on élimine de ce fait des configurations impossibles).

Une part de la complexité d'une fonction est donc liée au nombre total de configurations de connexions permises par l'outil SAGA. Nous ne prétendons pas que la complexité puisse être étudiée simplement sous cet angle ; d'autres facteurs peuvent influer sur cette complexité. Nous y reviendrons d'ailleurs ultérieurement.

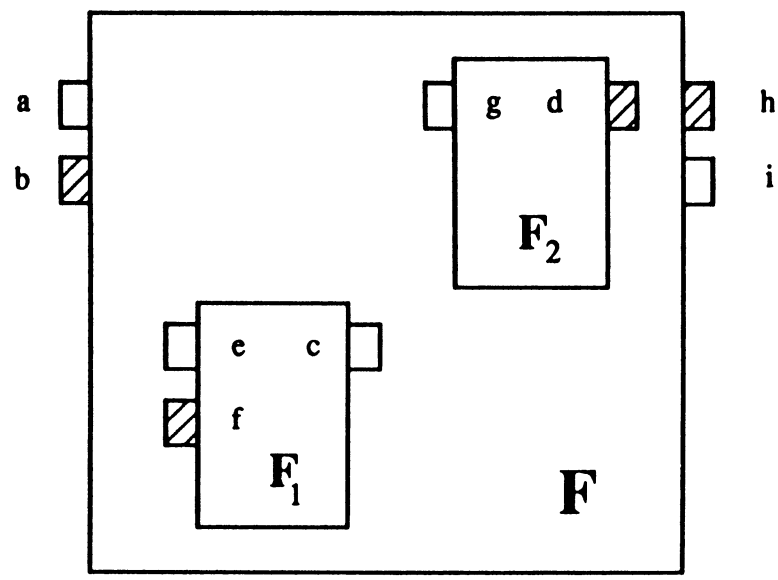
Nous avons quant à nous choisi de n'approfondir que cet aspect des choses. L'indicateur de complexité d'une fonction SAGA que nous avons retenu sera une fonction croissante du nombre de graphes de connexions possibles pour cette fonction. La métrique retenue présente le double mérite de correspondre à une certaine intuition de la complexité d'une fonction SAGA, et d'être calculable sans trop de difficultés mathématiques, son calcul étant en outre facilement automatisable.

L'utilisation de la théorie des graphes dans le domaine de la qualité de logiciels "flots de données" est également, à notre connaissance, une application nouvelle de cette théorie.

Dans le paragraphe suivant, on formalise le problème pour pouvoir ensuite calculer explicitement le nombre total de graphes de connexions associés à une fonction SAGA quelconque.

B) Formalisation

Reprenons la première configuration de l'exemple présenté ci-dessus :



(Figure 15) Fonction principale F

Nous allons commencer par définir les rôles des variables. La variable a, par exemple, est une variable d'entrée de la fonction F, comme i en est une variable de sortie. Le plot représentant la variable a va recueillir les différentes valeurs prises par cette variable au cours du temps, tout comme le plot représentant i va pouvoir fournir à d'autres variables les valeurs successives prises par i. De même, la variable e est une variable d'entrée de la fonction F_1. Mais si on connecte a à e, la première variable va se comporter en fournisseur d'information pour la seconde, donc comme la sortie d'une fonction imaginaire. On constate donc que le rôle d'entrée ou de sortie dévolu à une variable n'est valable qu'à l'intérieur d'une vue SAGA.

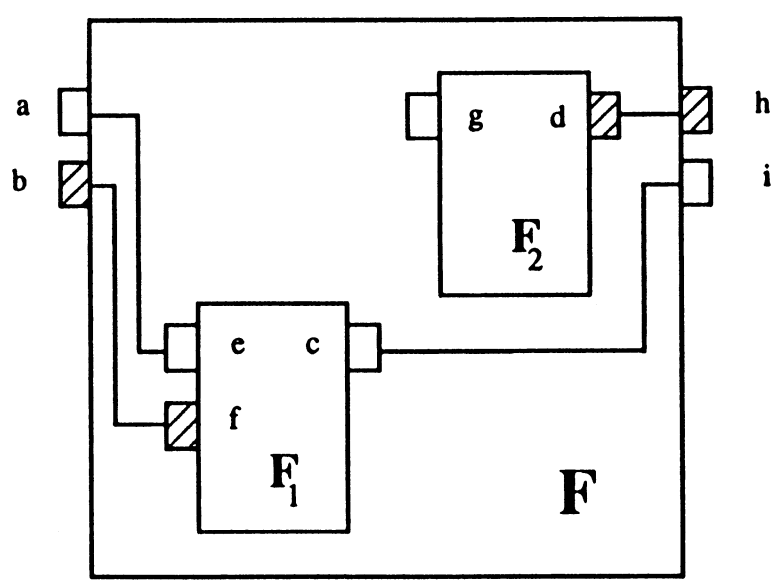
Pour une vue bien déterminée (il n'y a alors pas d'ambiguïté), on adoptera les termes de variables "productrices" de données (ou producteurs) comme a, b, c et d dans la figure 15, et de variables "consommatrices" de données (ou consommateurs) comme e, f, g, h et i.

Soient donc P_F le nombre de producteurs et C_F le nombre de consommateurs de la fonction F. Terminer la définition de la fonction F consiste maintenant à connecter l'ensemble des variables productrices aux variables consommatrices, sous plusieurs contraintes imposées par l'outil SAGA :

(C_1) on ne peut connecter deux variables que si celles-ci sont de même type ;
(C_2) on ne peut connecter deux variables que si celles-ci appartiennent à deux fonctions différentes (pas de bouclage sur une même fonction) ;
(C_3) chacun des producteurs doit être connecté à au moins un consommateur ;
(C_4) chacun des consommateurs doit être connecté à exactement un producteur.

Les conditions (C_3) et (C_4) indiquent que la relation qui à chaque consommateur associe un producteur est surjective, donc que $P_F \leq C_F$.

Sur la figure 15 par exemple, d'après (C_2), la variable e ne peut être connectée à c, donc il nous faut la connecter à a d'après (C_1). De même, h n'est pas connectable à b, donc nous devons la connecter à d. Il nous reste à connecter b à f, et i doit être connectée à c et pas à a, d'après (C_2). Nous obtenons alors le schéma incomplet suivant :



(Figure 16) Graphe de connexions incomplet

Comme présenté dans le paragraphe précédent, les configurations possibles des relations sur cette fonction sont donc au nombre de 2, suivant que l'on décide de connecter g à c ou g à a.

Dans le cas général, pour calculer le nombre de graphes de connexions possibles à partir de la représentation d'une fonction F, nous introduisons la matrice de connexions associée à cette fonction.

Pour ce faire, nous modélisons l'ensemble des connexions possibles entre variables d'une vue par une matrice $M = (M_{ij})_{(i=1,...,P\ ;\ j=1,...,C)}$ telle que $M_{ij} = 1$ s'il peut exister une connexion entre le producteur i et le consommateur j (on dit alors que la case [i, j] est autorisée, et que le producteur i et le consommateur j sont compatibles), et 0 sinon (la case [i, j] est alors qualifiée d'interdite), les critères choisis étant les contraintes (C_1) à (C_4) présentées ci-dessus.

Dans notre exemple, la matrice de connexions associée à F a la forme suivante :

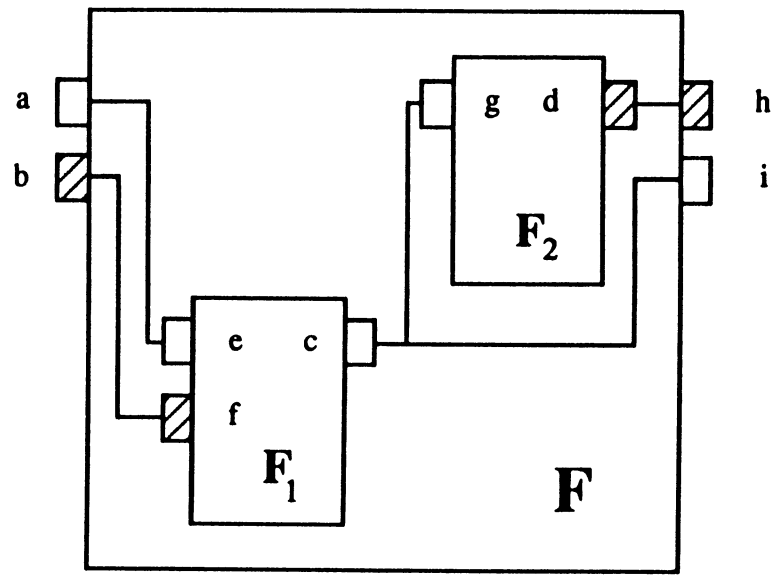
Consommateurs

P \ C	e	f	g	h	i
a	1	0	1	0	0
b	0	1	0	0	0
c	0	0	1	0	1
d	0	1	0	1	0

Producteurs

(Figure 17) Matrice de connexions associée à F

Si nous étudions les deux graphes de connexions possibles (on dit aussi réalisables) pour cette fonction en hachurant dans les matrices correspondantes les connexions choisies dans chaque cas, nous obtenons :



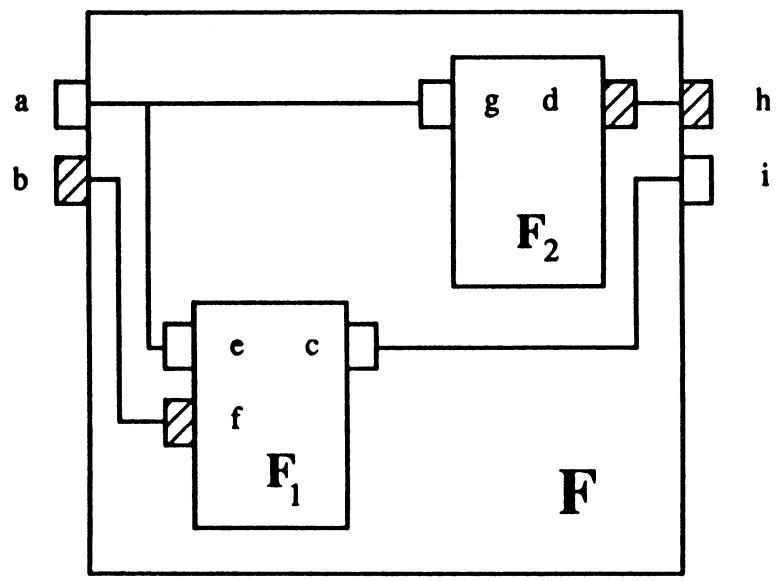
(Figure 18.1) Graphe 1

dont la matrice correspondante est

P \ C	e	f	g	h	i
a	1	0	1	0	0
b	0	1	0	0	0
c	0	0	1	0	1
d	0	1	0	1	0

(Figure 18.2) Matrice correspondant au graphe 1

et



(Figure 19.1) Graphe 2

dont la matrice correspondante est

P \ C	e	f	g	h	i
a	1	0	1	0	0
b	0	1	0	0	0
c	0	0	1	0	1
d	0	1	0	1	0

(Figure 19.2) Matrice correspondant au graphe 2

Choisir un graphe de connexions pour la fonction F revient à sélectionner des cases autorisées dans la matrice de connexions sous certaines contraintes ; ces contraintes apparaissent comme la transposition des contraintes concernant les variables. En effet :
(C_1) et (C_2) sont vérifiées par la définition correcte de la matrice de connexions ;
(C_3) implique que l'on choisisse au moins une case autorisée par ligne de la matrice ;
(C_4) implique qu'on sélectionne une et une seule case autorisée pour chaque colonne de la matrice.

Le problème se ramène donc à sélectionner C_F cases dans la matrice de connexions (une par colonne) de telle façon qu'au bout du compte, on en ait choisi au moins une par ligne.

Dans le paragraphe suivant, nous proposons une restructuration de la matrice de connexions, de manière à calculer plus facilement le nombre de graphes réalisables sur une fonction SAGA.

C) Calcul de la complexité d'assemblage d'une fonction SAGA

Nous avons vu que dans l'exemple précédent (figure 17), la matrice des connexions possibles s'écrivait :

Consommateurs

P \ C	e	f	g	h	i
a	1	0	1	0	0
b	0	1	0	0	0
c	0	0	1	0	1
d	0	1	0	1	0

Producteurs

(Figure 20) Matrice de connexions associée à la fonction F

Si nous réordonnons les lignes et les colonnes de cette matrice en regroupant côte à côte les variables de même type, nous obtenons :

P \ C	e	g	i	f	h
a	1	1	0	0	0
c	0	1	1	0	0
b	0	0	0	1	0
d	0	0	0	1	1

(Figure 21) Matrice de connexions réordonnée

Puisque d'après (C_1), on ne peut connecter deux variables que si celles-ci sont de même type, nous pouvons remarquer que le nombre total de graphes de connexions possibles dans une fonction est le produit des nombres de graphes possibles pour les variables de chacun des types représentés à l'intérieur de la fonction.

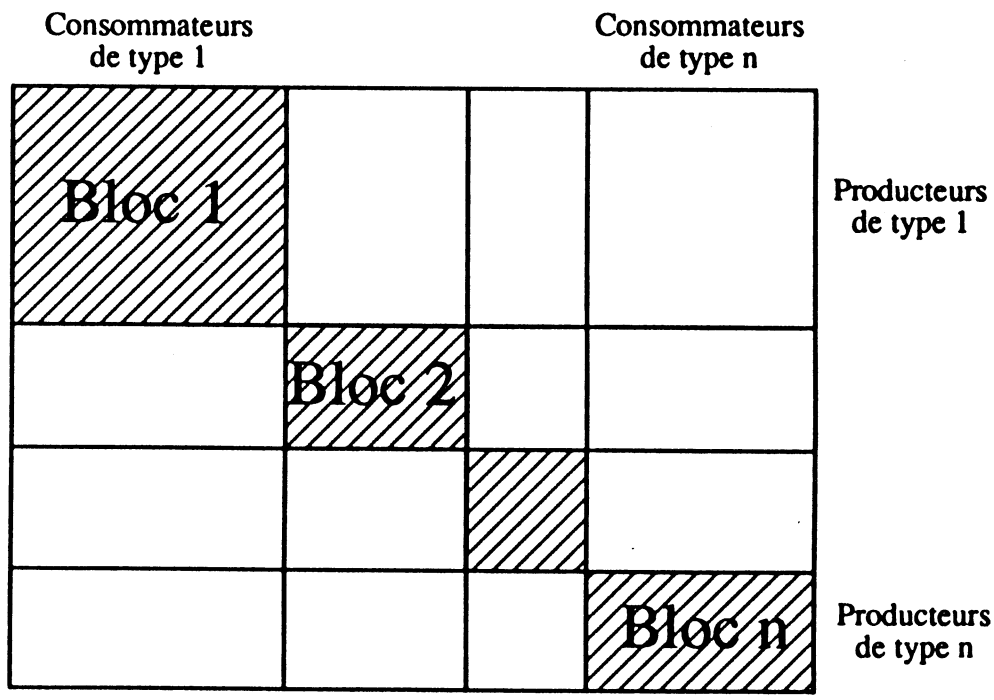
Définir un graphe de connexions pour une fonction SAGA revient ainsi à définir indépendamment les uns des autres les graphes de connexions des variables de même type de

cette fonction. C'est ce que nous retrouvons dans la matrice réordonnée ci-dessus : elle est diagonale par blocs de variables de même type (figure 22).

P \ C	e	g	i	f	h
a	1	1	0	0	0
c	0	1	1	0	0
b	0	0	0	1	0
d	0	0	0	1	1

(Figure 22) Aspect diagonal de la matrice de connexions de F

D'une manière générale, on pourra toujours regrouper les variables d'une même fonction par type, et obtenir une matrice de connexions réordonnée, diagonale par blocs (figure 23).



(Figure 23) Diagonalisation d'une matrice de connexions

Pour calculer le nombre total de graphes associés à une fonction F, il suffit de calculer le nombre de graphes possibles sur chacun des blocs de variables d'un même type, puis de faire le produit des résultats obtenus.

De même que nous avions permuté les lignes et les colonnes de la matrice pour obtenir des blocs de variables de même type, nous pouvons regrouper dans chacun des blocs de cette matrice les variables appartenant à une même fonction. Si nous procédons à cette opération sur notre exemple, nous obtenons la figure 24 suivante :

P \ C	g	i	e	f	h
a	1	0	1	0	0
c	1	1	0	0	0
d	0	0	0	1	1
b	0	0	0	1	0

(Figure 24) Matrice de connexions réordonnée

Nous constatons que chacun des deux blocs de la matrice apparaît à son tour pseudo-diagonal par sous-blocs nuls, ces sous-blocs nuls correspondant à des variables appartenant à une même sous-fonction.

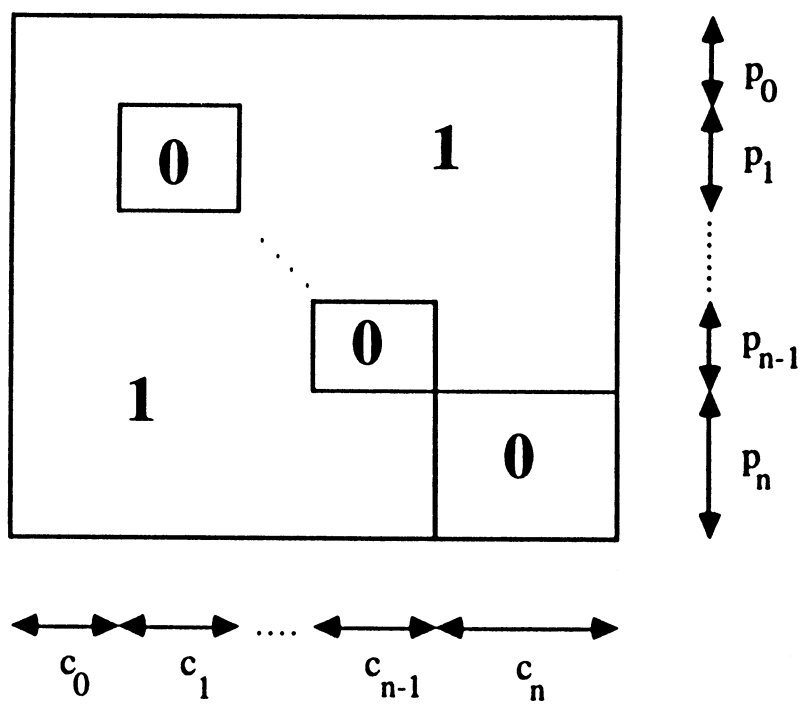
Dans l'exemple de la figure 24, les variables e et c appartiennent toutes deux à la fonction F_1 (figure 15), a et i à la fonction principale F, et b et h également à F. On dit alors que e et c appartiennent à la même famille de variables (la case correspondante dans la matrice de connexions est alors nulle d'après (C_2)) ; d'une manière plus générale, on peut grouper avec e tous les consommateurs de type 1 appartenant à la fonction F_1 et avec c tous les producteurs de type 1 appartenant à la fonction F_1.

Deux variables d'une même vue SAGA appartiennent donc à la même famille si et seulement si les deux conditions suivantes sont remplies simultanément :
- elles sont de même type (elles appartiennent alors au même bloc) ;
- elles appartiennent à la même fonction dans la vue (ce peut être la fonction principale).

Quant à la variable d, nous constatons qu'elle peut être connectée à tous les consommateurs du même type qu'elle dans la vue, puisque d est la seule variable productrice de type 2 de la sous-fonction F_2 ; elle n'appartient donc à aucune famille : on dit que d est un producteur libre. De même, f et g sont des consommateurs libres.

On dit qu'un producteur (resp. consommateur) est libre s'il n'existe pas dans la vue correspondante de consommateur (resp. producteur) de même type appartenant à la même fonction.

De manière générale, on montre facilement que tout bloc de connexions peut se mettre par permutation des lignes et des colonnes sous la forme suivante :



(Figure 25) Allure générale d'un bloc de matrice de connexions

$\forall\, i \in \{1,\ldots,n\}$, les producteurs correspondant aux lignes repérées par p_i et les consommateurs correspondant aux colonnes repérées par c_i appartiennent à la même famille.

Les producteurs (resp. consommateurs) correspondant aux lignes repérées par p_0 (resp. c_0) sont des producteurs libres (resp. consommateurs libres).

Sur ce bloc, toutes les variables représentées sont de même type, et l'on a $\sum_{i=0}^{n} c_i = C$ (nombre total de consommateurs du bloc) et $\sum_{i=0}^{n} p_i = P$ (nombre total de producteurs du bloc).

n est le nombre de familles du bloc, c'est-à-dire le nombre de fonctions de la vue contenant simultanément des producteurs et des consommateurs de même type.

On peut noter que la définition du problème ressemble à celle des cases admissibles, sujet traité dans [SAK-84, p.94] ; néanmoins, les contraintes imposées ici sont trop différentes pour pouvoir appliquer les résultats de M. Sakarovitch.

Il nous reste donc à calculer le nombre de graphes possibles correspondant à un bloc donné.

On notera $N(c_0,p_0,c_1,p_1,\ldots,c_n,p_n)$ le nombre de graphes de connexions réalisables à partir d'un bloc de connexions du type précédent.

Remarque : dans cette formulation, pour une valeur de n fixée strictement positive, il est nécessaire que les valeurs de $c_1,\ldots,c_n$ et de $p_1,\ldots,p_n$ soient strictement positives ; seuls c_0 et p_0 peuvent être éventuellement nuls (cas où il n'existe pas pour le bloc de consommateurs libres ou de producteurs libres).

Nous allons montrer dans les paragraphes suivants qu'à l'aide de ce formalisme, on peut calculer le nombre de graphes possibles pour chacun des blocs extraits d'une vue SAGA quelconque.

Dans un premier temps, nous allons déterminer une condition nécessaire et suffisante pour qu'il existe au moins un graphe de connexions possible réalisable associé à un bloc $(c_0,p_0,c_1,p_1,\ldots,c_n,p_n)$ donné ; on dira alors que le bloc est réalisable. Puis nous nous intéresserons au calcul du nombre de graphes réalisables dans le cas où le bloc ne comporte que des variables libres (n=0, bloc simple) ; enfin nous étudierons le cas général.

C-1) Bloc réalisable

On dira qu'une matrice de connexions est réalisable si l'on peut définir au moins un graphe de connexions réalisable sur la fonction correspondante.

Trivialement, une matrice de connexions est réalisable si et seulement si chacun des blocs de connexions qui la composent est lui-même réalisable. Nous étudions dans ce paragraphe les conditions sous lesquelles un bloc de connexions est réalisable.

Nous avons déjà vu que pour pouvoir définir au moins un graphe réalisable à partir de la matrice d'une fonction F, il faut $P_F \leq C_F$ d'après (C_3) et (C_4) (au moins autant de consommateurs que de producteurs). Il est clair que cette relation doit être vraie également pour chacun des types de variables en présence dans la fonction, puisqu'un graphe de connexions peut être construit type par type ; pour qu'un bloc B quelconque de la matrice soit réalisable, il faut donc $P \leq C$.

Le problème que nous nous posons ici est de trouver une condition nécessaire et suffisante pour qu'à un bloc $B = (c_0,p_0,c_1,p_1,\ldots,c_n,p_n)$ donné, on puisse associer au moins un graphe réalisable. Tout graphe réalisable associé au bloc B constitue alors une affectation producteurs-consommateurs sur ce bloc.

Dans toute la suite de ce document, nous désignerons un producteur par la lettre π et un consommateur par la lettre χ.

Lemme : sous l'hypothèse $P \leq C$, le bloc B est réalisable si et seulement si on peut trouver un sous-ensemble de P consommateurs tel que chacun de ces P consommateurs soit connectable à un producteur différent.

Preuve Condition nécessaire : si le bloc B est réalisable, alors il existe une affectation sur ce bloc, donc en particulier on peut trouver un sous-ensemble de P consommateurs associés chacun à un producteur différent.

Condition suffisante : si un tel sous-ensemble existe, alors tous les producteurs sont connectés, et la contrainte (C_3) est vérifiée ; pour exhiber un graphe réalisable il suffit de vérifier (C_4), donc de connecter les C-P consommateurs restants chacun avec un producteur, en tenant compte des cases interdites du bloc de connexions. Le graphe obtenu est alors bien réalisable.

Nous énonçons maintenant une condition nécessaire et suffisante à l'existence d'une affectation.

Théorème : sous l'hypothèse $P \leq C$, on a :
- le bloc B est toujours réalisable pour $n = 0$;
- le bloc B est réalisable pour $n > 0$ si et seulement si $\forall i \in \{1,2,\ldots,n\}$, $p_i + c_i \leq C$.

Pour qu'un bloc B soit réalisable avec $n > 0$, il faut donc et il suffit que les p_i producteurs de chaque famille i, pris indépendamment des autres producteurs, puissent être connectés chacun au moins à un consommateur (il y a $C-c_i$ consommateurs disponibles pour ces producteurs), soit $p_i \leq C - c_i$.

Preuve si $n = 0$: on peut facilement exhiber une affectation réalisable sur B, en sélectionnant les P premiers consommateurs et en les associant aux P premiers producteurs dans cet ordre. On est alors dans les hypothèses du lemme précédent.

$n > 0$: (Condition nécessaire) Supposons le bloc B réalisable avec $n > 0$, et considérons une affectation sur B ; soit alors i une famille quelconque de B ($i \in \{1,2,\ldots,n\}$) : cette famille comporte c_i consommateurs et p_i producteurs. Comme tous les producteurs de la vue sont connectés chacun au moins à un consommateur dans l'affectation d'après (C_3), c'est aussi vrai pour les p_i producteurs de la famille i ; mais ceux-ci n'ont pu être connectés qu'à des consommateurs n'appartenant pas à la famille i (il y a en tout $C - c_i$ consommateurs de ce type) ; il vient donc $C - c_i \geq p_i$.

Donc $\forall i \in \{1,2,\ldots,n\}$, $c_i + p_i \leq C$.

(Condition suffisante) On suppose maintenant que le bloc B vérifie $\forall i \in \{1,2,\ldots,n\}$, $p_i + c_i \leq C$, avec $n > 0$; montrons qu'alors il existe au moins une affectation sur B. Considérons pour cela le graphe biparti $G = (\pi, \chi, E)$, où $\pi = \{\pi_1,\pi_2,\ldots,\pi_P\}$ est l'ensemble des producteurs du bloc B, $\chi = \{\chi_1,\chi_2,\ldots,\chi_C\}$ celui des consommateurs, et E celui des arêtes entre les éléments compatibles de ces ensembles : on a une arête (π_i, χ_j) si et seulement si $M_{ij} = 1$.

Pour prouver que le bloc B est réalisable, il suffit d'après le lemme précédent d'exhiber un sous-ensemble de P consommateurs associés chacun à un producteur différent, c'est-à-dire un couplage de cardinalité P dans le graphe G (on rappelle qu'un ensemble Y d'arêtes est un couplage si et seulement si deux arêtes quelconques de cet ensemble n'ont aucune extrémité commune).

Si X est un sous-ensemble de π (ou de χ), on note $|X|$ le cardinal de X et V(X) l'ensemble des points de χ qui sont adjacents à au moins l'un des points de X.

Montrons que sous les hypothèses $p_i + c_i \leq C \ \forall i \in \{1,2,\ldots,n\}$, il existe un couplage de π dans χ. Pour ce faire, on utilise le théorème suivant [BER-83, p.128] :

Théorème de Hall (1934) : soit $G = (\pi, \chi, E)$ un graphe biparti ; alors G possède un couplage saturant les sommets de π si et seulement si $\forall X, \pi \supseteq X, |X| \leq |V(X)|$.

Dans notre cas, soit X un ensemble quelconque non vide de producteurs (sous-ensemble de π). Alors :

- soit X ne possède que des producteurs libres, auquel cas $|X| \leq p_0 \leq C = |V(X)|$;
- soit X ne possède que des producteurs d'une certaine famille i, auquel cas
$$|X| \leq p_i \leq C - c_i = |V(X)| ;$$
- soit X possède au moins deux producteurs de familles différentes (ou un producteur appartenant à une famille et un producteur libre), auquel cas on a $|X| \leq P \leq C = |V(X)|$.

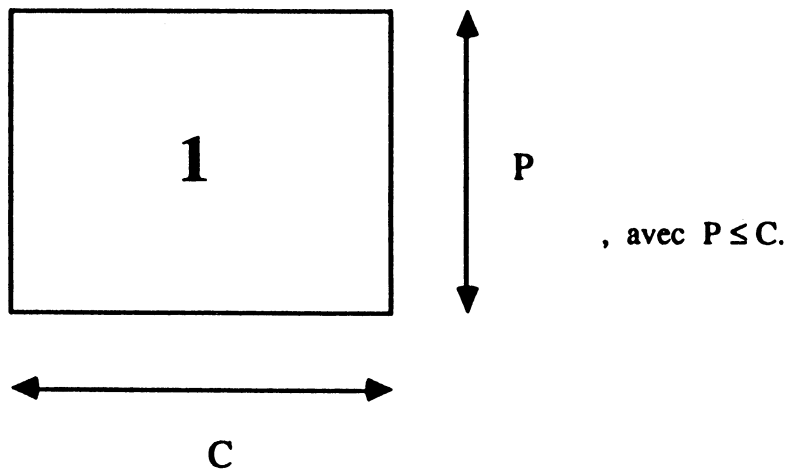
Il existe donc bien un couplage de π dans χ, donc le bloc B est effectivement réalisable.

Je tiens ici à remercier MM. Olivier BAUDON et Claude BENZAKEN, du Laboratoire de Structures Discrètes de Grenoble, qui m'ont aidé pour la démonstration de ce théorème.

Nous présentons maintenant la méthode de calcul du nombre de graphes associé à un bloc extrait d'une vue SAGA.

C-2) Calcul pour un bloc simple

Nous considérons dans ce paragraphe le cas n = 1 ; tous les producteurs et les consommateurs du type considéré dans le bloc sont alors des variables libres, et peuvent être connectés entre eux. Le bloc de connexions a alors l'allure suivante :



(Figure 26) Bloc de connexions simple

Nous avons vu que choisir un graphe de connexions revient à sélectionner une et une seule case par colonne (connexion de chacun des consommateurs), de manière à avoir choisi au bout du compte au moins une case par ligne (tous les producteurs doivent être connectés).

On peut faire l'analogie avec le problème consistant à répartir C boules numérotées dans P casiers numérotés, de telle manière que chacun des casiers reçoive au moins une boule.

Si les casiers étaient indiscernables, le nombre total d'affectations possibles serait donné par le nombre de Stirling de deuxième espèce S_C^P, défini comme le nombre de façons d'agencer une collection de C objets différents dans un ensemble de P boîtes indiscernables, en ne laissant aucune boîte vide. D'après [BER-68, p.35], on a :

$$S_{C+1}^P = S_C^{P-1} + P \,.\, S_C^P \quad \text{pour} \quad 1 < P < C \,, \quad \text{et} \quad S_C^1 = S_C^C = 1.$$

Ici, on a affaire à des casiers numérotés (les producteurs sont discernables) ; ce résultat est donc à multiplier par le nombre de façons d'agencer les P casiers entre eux, soit P ! . On obtient alors le nombre de surjections de l'ensemble { 1, 2, ... , C } dans l'ensemble { 1, 2, ... , P } , soit [BER-68, p.36] :

$$N(C,P) = P\,! \,.\, S_C^P$$

Nous avons alors un moyen très simple pour calculer de proche en proche les nombres N(C,P) ; en effet, on a :

$$N(C,P) = P! \,.\, S_C^P \quad \text{et} \quad N(C,P-1) = (P-1)! \,.\, S_C^{P-1}.$$

$$N(C,P) = P! \,.\, S_C^P \quad \text{et} \quad N(C,P-1) = (P-1)! \,.\, S_C^{P-1}.$$

Donc $N(C,P) + N(C,P-1) = (P-1)! \,.\, [\, S_C^{P-1} + P \,.\, S_C^P \,] = (P-1)! \,.\, S_{C+1}^P$,
et par suite $N(C+1,P) = P \,.\, [\, N(C,P) + N(C,P-1) \,]$.

On a aussi $N(C,1) = S_C^1 = 1$, et $N(C,C) = C! \,.\, S_C^C = C!$. Donc :

Relation de récurrence 1 :

$N(C,1) = 1$ pour $C \geq 1$;

$N(C,C) = C!$ pour $C \geq 1$;

$N(C+1,P) = P \,.\, [\, N(C,P) + N(C,P-1) \,]$ pour $1 < P < C$.

A partir de ces constatations, on peut calculer le nombre d'affectations possibles pour les premières valeurs de C et de P, de proche en proche. Si on classe les résultats, on obtient le tableau suivant :

C \ P	1	2	3	4	5	6	7	8	
1	1								
2	1	2							
3	1	6	6						
4	1	14	36	24					
5	1	30	150	240	120				
6	1	62	540	1560	1800	720			
7	1	126	1806	8400	16800	15120	5040		
8	1	254	5796	40824	126000	191520	141120	40320	

(Figure 27) Tableau des valeurs de N(C,P) pour un bloc simple

Si l'on exprime maintenant N(C-1,P) par la formule énoncée plus haut, on a

$$N(C-1,P) = P \,.\, [\, N(C-2,P) + N(C-2,P-1) \,] \,,$$

$N(C,P) = P \cdot N(C-1,P-1) + P^2 \cdot N(C-2,P-1) + P^3 \cdot N(C-3,P-1) + P^3\, N(C-3,P)$, et ainsi de suite.

La dernière étape de cette décomposition est :

$N(C,P) = P \cdot N(C-1,P-1) + P^2 \cdot N(C-2,P-1) + \ldots + P^{C-P} \cdot N(P,P-1) + P^{C-P}\, N(P,P)$;

or $N(P,P) = P \cdot N(P-1,P-1)$. On a donc :

$$N(C,P) = \sum_{k=1}^{C-P+1} P^k \cdot N(C-k,P-1) \quad , \text{ soit } \quad N(C,P) = \sum_{j=P-1}^{C-1} P^{C-j} \cdot N(j,P-1) .$$

Nous avons donc une seconde relation :

Relation de récurrence 2 :

$$N(C,1) = 1 \qquad \text{pour } C \geq 1 ;$$

$$N(C,P) = \sum_{j=P-1}^{C-1} P^{C-j} \cdot N(j,P-1) \qquad \text{pour } 1 < P \leq C.$$

Considérons maintenant la décomposition du problème suivante : pour $1 < P \leq C$, soit k le nombre d'objets contenus dans le premier casier.

On a $1 \leq k \leq C-P+1$: en effet, il est nécessaire que les P-1 casiers restants contiennent chacun au moins une boule ; il ne reste donc plus que $C-(P-1) = C-P+1$ boules à la disposition du premier casier.

Supposons donc que le casier numéro 1 contienne k boules, $k \in \{ 1, 2, \ldots, C-P+1 \}$ (il y a C_C^k façons de choisir ces k boules) ; le problème se ramène alors à dénombrer le nombre de façons de ranger les C-k boules distinctes restantes parmi les P-1 derniers casiers sans qu'aucun casier ne se retrouve vide. On a donc :

$$N(C,P) = \sum_{k=1}^{C-P+1} C_C^k \cdot N(C-k,P-1) \quad , \text{ soit par le changement de variable } j = C - k$$

$$N(C,P) = \sum_{j=P-1}^{C-1} C_C^j \cdot N(j,P-1) .$$

De plus, comme précédemment, on a $N(C,1) = 1$ pour tout $C \geq 1$. D'où la relation :

Relation de récurrence 3 :

$$N(C,1) = 1 \qquad \text{pour } C \geq 1 ;$$

$$N(C,P) = \sum_{j=P-1}^{C-1} C_C^j \cdot N(j,P-1) \text{ pour } 1 < P \leq C.$$

D'autre part, d'après [BER-68, p.66], on a pour tout couple (C,P) tel que $1 \leq P \leq C$:

$$S_C^P = \frac{1}{P!} \cdot \sum_{k=0}^{P} (-1)^{P-k} \cdot C_P^k \cdot k^C .$$

On en déduit l'expression directe suivante des nombres N(C,P) :

Relation 4 :

$$N(C,P) = \sum_{k=0}^{P} (-1)^{P-k} \cdot C_P^k \cdot k^C \text{ pour } 1 \leq P \leq C.$$

Nous avons donc à notre disposition quatre moyens différents de calculer le nombre de graphes correspondant à un bloc simple. Il est important de bien maîtriser ces premiers calculs, puisque dans le cas général, on procédera à une analyse récursive qui s'achevera par un de ces calculs.

C-3) Calcul pour un bloc quelconque

On dispose ici d'un bloc défini par $(c_0,p_0,c_1,p_1,\ldots,c_n,p_n)$, avec $n \geq 1$.Pour calculer le nombre de graphes de connexions possibles sur ce bloc, nous allons d'abord classer l'ensemble de ces graphes : soit E_i l'ensemble des graphes réalisables sur le bloc $(c_0,p_0,c_1,p_1,\ldots,c_n,p_n)$ tels que le dernier producteur du bloc, soit π_P, soit connecté simultanément à i consommateurs. On a alors facilement :

$$N(c_0,p_0,c_1,p_1,\ldots,c_n,p_n) = \sum_{i=1}^{C} |E_i| ,$$

où $|E_i|$ représente le cardinal de l'ensemble E_i. On peut d'ailleurs tronquer cette expression en remarquant que si i consommateurs sont connectés à π_P, il reste alors C-i consommateurs pour P-1 producteurs. Comme précédemment, nous constatons que pour construire un graphe réalisable, il faut encore pouvoir connecter les variables restantes entre elles, soit nécessairement d'après (C_3), $C-i \geq P-1$, ou encore $i \leq C-P+1$. On a donc :

$$N(c_0,p_0,c_1,p_1,\ldots,c_n,p_n) = \sum_{i=1}^{C-P+1} |E_i| .$$

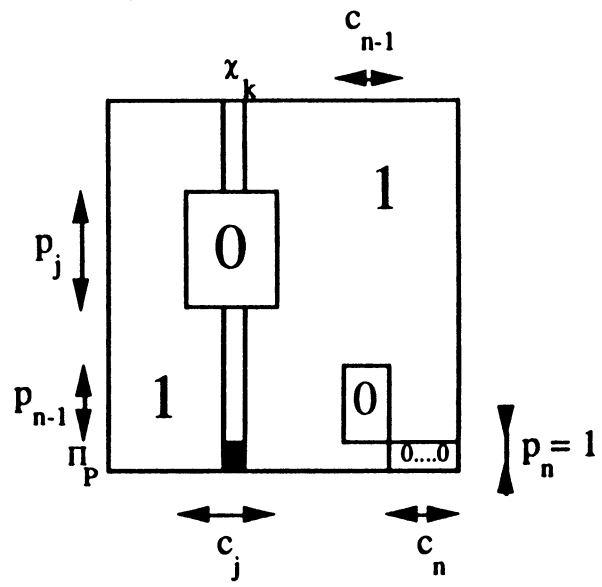
Dans la suite de ce chapitre, nous proposons une méthode de calcul pour $|E_i|$. On commence par le cas $i = 1$, puis on présente le calcul général.

C-3-1) Calcul de $|E_1|$

Nous nous proposons donc ici de calculer le nombre de graphes réalisables à partir du bloc $(c_0,p_0,c_1,p_1,\ldots,c_n,p_n)$ et comportant un seul consommateur connecté au producteur π_P. Soit χ_k ce consommateur, et soit j le numéro de la famille à laquelle il appartient (éventuellement $j = 0$ si le consommateur est libre).

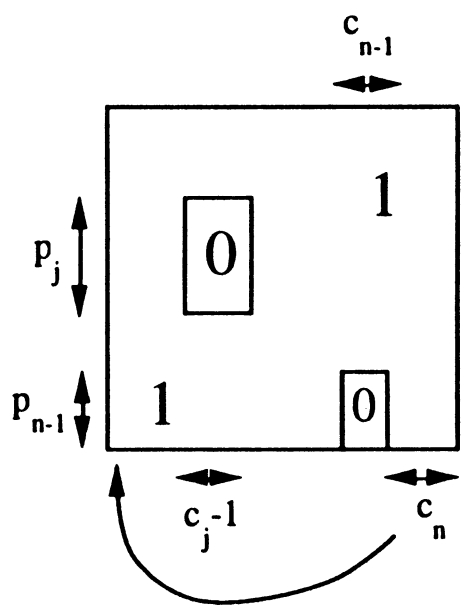
On a donc $c_j \geq 1$, et l'une des variables de la j^e famille, χ_k , est le seul consommateur connecté à π_P. Deux cas sont alors possibles :

- $p_n = 1$; on a la figure 28 suivante :



(Figure 28) Allure du bloc avant connexion

Après connexion de π_P et χ_k, on n'a alors plus aucun producteur du type considéré appartenant à la $n^{ième}$ famille. Dans tous les graphes de connexions du type cherché, les c_n consommateurs de la $n^{ième}$ famille sont donc connectés à n'importe lesquels des producteurs restants ; on peut alors considérer pour la suite du calcul qu'ils viennent agrandir la famille des consommateurs libres. On a alors la figure 29 suivante :



(Figure 29) Allure du bloc après connexion

Calculer $|E_1|$ revient dans ce cas à sommer pour chaque consommateur connectable à π_P le nombre de graphes de connexions possibles sur les variables restantes ; on obtient :

$$|E_1| = \sum_{j=1}^{n-1} c_j * N(c_0+c_n,p_0,c_1,p_1,\ldots,c_{j-1},p_{j-1},c_j-1,p_j,c_{j+1},p_{j+1},\ldots,c_{n-1},p_{n-1}) + c_0 * N(c_0-1,p_0,c_1,p_1,\ldots,c_{n-1},p_{n-1}) .$$

Le terme $c_0 * N(c_0-1,p_0,c_1,p_1,\ldots,c_{n-1},p_{n-1})$ exprime le nombre de graphes réalisables par connexion de π_P à un consommateur libre.

On appelle bloc équivalent restant le bloc de connexions constitué par la matrice de connexions des variables restant à connecter après connexion du producteur π_P. Nous verrons par la suite comment définir ce bloc avec le formalisme du paragraphe C.

- $p_n \geq 2$; on raisonne de la même manière que précédemment, mais après connexion de π_P et χ_k, la $n^{ième}$ famille n'est pas éliminée, et comporte encore p_n-1 producteurs et c_n consommateurs à connecter. On obtient donc :

$$|E_1| = \sum_{j=1}^{n-1} c_j * N(c_0,p_0,c_1,p_1,\ldots,c_{j-1},p_{j-1},c_j-1,p_j,c_{j+1},p_{j+1},\ldots,c_{n-1},p_{n-1},c_n,p_n-1) + c_0 * N(c_0-1,p_0,c_1,p_1,\ldots,c_{n-1},p_{n-1},c_n,p_n-1) .$$

C-3-2) Calcul de $|E_j|$ avec $j>1$

On suppose maintenant que j consommateurs sont connectés à π_P, avec $j > 1$. Soient donc γ_0 le nombre de consommateurs libres connectés à π_P ($\gamma_0 \leq c_0$), γ_1 le nombre de consommateurs de la première famille connectés à π_P ($\gamma_1 \leq c_1$),..., γ_{n-1} le nombre de consommateurs de la $(n-1)^e$ famille connectés à π_P ($\gamma_{n-1} \leq c_{n-1}$).

On a $j = \gamma_0+\gamma_1+\ldots+\gamma_{n-1}$; c'est le nombre total de consommateurs connectés à π_P. On peut alors considérer le j-uplet des consommateurs connectés à π_P et dont l'appartenance aux diverses familles est déterminée par $\gamma = (\gamma_0,\gamma_1,\ldots,\gamma_{n-1})$; ce dernier vecteur est appelé vecteur de connexions.

Le nombre de j-uplets de consommateurs ayant le vecteur de connexions $\gamma = (\gamma_0,\gamma_1,\ldots,\gamma_{n-1})$ est $C_{c_0}^{\gamma_0} . C_{c_1}^{\gamma_1} \ldots C_{c_{n-1}}^{\gamma_{n-1}} = \prod_{k=0}^{n-1} C_{c_k}^{\gamma_k}$.

Calculer $|E_j|$ revient donc dans ce cas à sommer pour chaque vecteur de connexions possible sur le bloc et pour tous les j-uplets correspondants, le nombre de graphes de connexions possibles sur le bloc équivalent restant ; on obtient :

$$|E_j| = \sum_{(\gamma_0,\ldots,\gamma_{n-1}) \in \prod_{i=0}^{n-1} \{0,1,\ldots,c_i\},\ \sum_{i=0}^{n-1} \gamma_i = j} \left(\prod_{k=0}^{n-1} C_{c_k}^{\gamma_k}\right) . N'(c_0-\gamma_0,p_0,c_1-\gamma_1,p_1,\ldots,c_{n-1}-\gamma_{n-1},p_{n-1},c_n,p_n-1),$$

où $N'(c_0-\gamma_0,p_0,c_1-\gamma_1,p_1,\ldots,c_{n-1}-\gamma_{n-1},p_{n-1},c_n,p_n-1)$ désigne le nombre de graphes réalisables sur le bloc équivalent restant.

Dans le paragraphe suivant, on montre comment déterminer ce bloc équivalent restant.

C-3-3) Détermination du bloc équivalent

Dans cette partie, nous définissons à l'aide du formalisme du paragraphe II-3 le bloc équivalent restant une fois le vecteur de connexions $\gamma = (\gamma_0,\gamma_1,\ldots,\gamma_{n-1})$ choisi, et les connexions correspondantes effectuées pour π_P.

La définition initiale du bloc est $(c_0,p_0,c_1,p_1,\ldots,c_n,p_n)$. Considérons les modifications apportées par les connexions de π_P à chacune des familles :

- les familles i telles que $\gamma_i = c_i$, avec $0 \leq i \leq n-1$; tous les consommateurs concernés sont connectés à π_P, et les producteurs de la même famille deviennent libres ;
- les familles i telles que $\gamma_i < c_i$, avec $0 \leq i \leq n-1$; il leur reste $c_i-\gamma_i$ consommateurs, et p_i producteurs ;
- la famille n : si $p_n = 1$, alors il ne reste plus de producteurs dans cette famille, et les consommateurs correspondants sont libres ; sinon, il reste c_n consommateurs et p_n-1 producteurs.

Soit donc $N(c'_0,p'_0,c'_1,p'_1,\ldots,c'_{n'},p'_{n'})$ le nombre de graphes de connexions possibles sur les variables restantes après connexion des consommateurs choisis au producteur π_P.

Pour déterminer le nouveau bloc $B' = (c'_0,p'_0,c'_1,p'_1,\ldots,c'_{n'},p'_{n'})$ en fonction du bloc initial $B = (c_0,p_0,c_1,p_1,\ldots,c_n,p_n)$ et du vecteur de connexions $\gamma = (\gamma_0,\gamma_1,\ldots,\gamma_{n-1})$, on applique l'algorithme pseudo-pascal suivant :

Algorithme de détermination du bloc équivalent

Entrée : Bloc initial B ;
Vecteur de connexions γ ;

Sortie : Bloc équivalent B' ;

début_algorithme

$c'_0 = c_0 - \gamma_0$; (* Connexion des producteurs libres à π_P *)
$p'_0 = p_0$; (* Nombre initial de producteurs libres *)
$n' = 0$; (* Numéro de la nouvelle famille représentée *)
Pour i de 1 à n-1 faire
    si $\gamma_i = c_i$ alors (* On rend libres les producteurs correspondants *)
        $p'_0 = p'_0 + p_i$
    sinon (* On met à jour le nombre de consommateurs correspondant *)
        $n' = n' + 1$;
        $c'_{n'} = c_i - \gamma_i$;
        $p'_{n'} = p_i$
    finsi
finpour ;
si $p_n = 1$ alors
    $c'_0 = c'_0 + c_n$
sinon
    $n' = n' + 1$;
    $p'_{n'} = p_n - 1$
finsi

fin_algorithme.

Nous pouvons maintenant écrire la formule donnant le nombre de graphes réalisables à partir du bloc initial, de façon récursive :

$$N(c_0,p_0,c_1,p_1,\ldots,c_n,p_n) = \sum_{m=1}^{C-P+1} \sum_{(\gamma_0,\ldots,\gamma_{n-1})\in \prod_{i=0}^{n-1}\{0,1,\ldots,c_i\},\ \sum_{i=0}^{n-1}\gamma_i=m} \left(\prod_{j=0}^{n-1} C_{c_j}^{\gamma_j}\right) . N'(c_0-\gamma_0,p_0,\ldots,c_{n-1}-\gamma_{n-1},p_{n-1},c_n,p_n-1) ,$$

où $N'(c_0-\gamma_0,p_0,\ldots,c_{n-1}-\gamma_{n-1},p_{n-1},c_n,p_n-1)$ désigne le nombre de graphes réalisables sur le bloc équivalent restant après connexion des consommateurs sélectionnés par le vecteur γ et connectés à π_P.

La récursion s'arrête lorsque la détermination du bloc équivalent conduit à un bloc non réalisable (le nombre de graphes associés est alors égal à 0), ou à un graphe simple (dans ce cas n=0, et on se ramène aux calculs du paragraphe C-2).

En effet, le bloc équivalent après connexion du producteur π_P contient P-1 producteurs, C-x consommateurs (si x sont connectés à π_P, $x \geq 1$), et au maximum n familles de variables. Mais le nombre de consommateurs pour chaque famille est inférieur ou égal à celui des consommateurs correspondants dans le bloc initial, avec l'inégalité stricte pour au moins une des familles ou pour les consommateurs libres (puisque le producteur π_P est connecté). Donc, en un nombre fini (inférieur ou égal à C) d'étapes, le bloc équivalent final ne comportera plus que des variables libres.

Pour éclairer ce raisonnement, nous présentons ci-après l'application de cet algorithme sur deux exemples concrets.

C-4) Exemples

C-4-1) Exemple 1

Reprenons l'exemple de la figure 15 présentée en B. La matrice réordonnée a alors la forme suivante (cf. figure 24) :

	g	i	e	f	h
a	1	0	1	0	0
c	1	1	0	0	0
d	0	0	0	1	1
b	0	0	0	1	0

(Figure 30) Matrice de connexions réordonnée

Cette matrice est donc formée de deux blocs de variables de types différents, soient :

	f	h
d	1	1
b	1	0

	g	i	e
a	1	0	1
c	1	1	0

(Figure 31) Bloc B_1 Bloc B_2

Le nombre total de graphes de connexions possibles sur cette fonction est donc le produit des nombres de graphes de connexions par blocs, soit $N(F,F_1,F_2) = N(B_1) . N(B_2)$.

Calcul du nombre de graphes de connexions sur B_1 :

Nous cherchons N(1,1,1,1), avec n = 1 et C = P = 2, donc la seule valeur possible pour j est j = C-P+1 = 1 (un seul consommateur connectable à b). De plus, le seul consommateur connectable à b est f d'après la matrice. On a donc $\gamma_0 = 1$, d'où le bloc équivalent restant (1,1) (figure 32).

	h
d	**1**

(Figure 32) Bloc équivalent restant pour j = 1 et $\gamma = 1$

N(1,1) = 1, donc $N(B_1) = 1$.

Calcul du nombre de graphes de connexions sur B_2 :

Nous cherchons ici N(1,0,1,1,1,1) avec n = 2. On a P = 2 et C = 3. On peut alors avoir j = 1 ou j = 2.

* Pour j = 1, on vérifie que les seules valeurs possibles du vecteur de connexions sont γ = (1,0) ou γ = (0,1).

1^er cas) γ = (1,0) : par application de l'algorithme ci-dessus, le bloc équivalent restant est alors (1,0,1,1) (figure 33).

	e	i
a	**1**	**0**

(Figure 33) Bloc équivalent restant pour j = 1 et γ = (1,0)

Aucun graphe de connexions ne peut être associé à cette matrice, puisqu'un des deux consommateurs ne pourra jamais être connecté ; le nombre correspondant de graphes est donc nul ;

2ème cas) γ = (0,1) : le bloc équivalent restant est alors (2,1), et N(2,1) = 1 (figure 34).

	g	e
a	1	1

(Figure 34) Bloc équivalent restant pour j = 1 et $\gamma = (0,1)$

* Pour j = 2, la seule valeur possible pour le vecteur de connexions est $\gamma = (1,1)$; le bloc équivalent restant est alors (1,1)

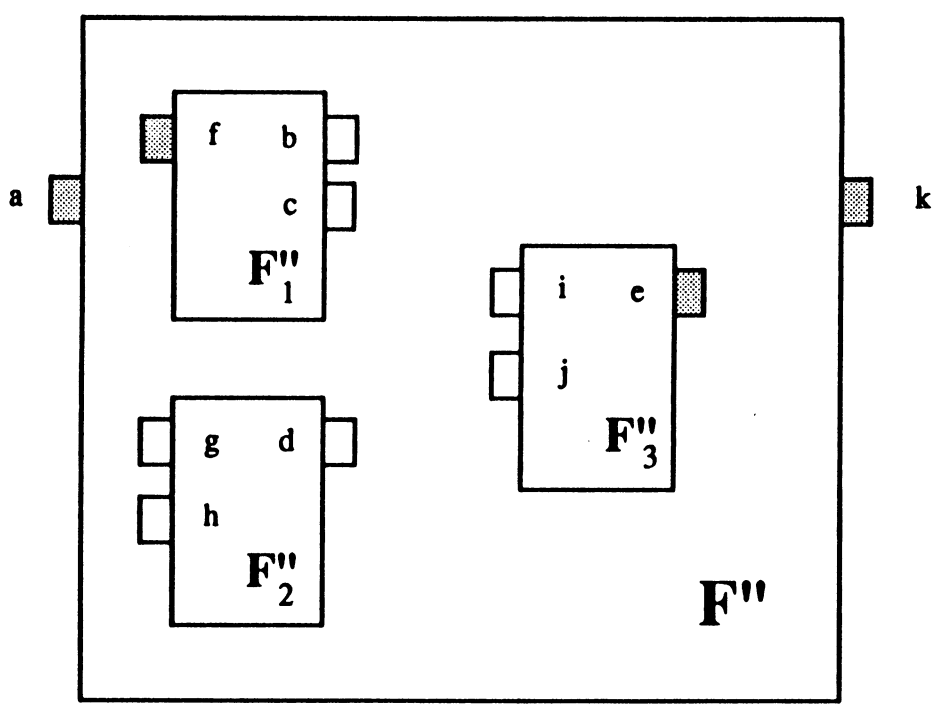
	e
a	1

, et N(1,1) = 1.

Donc $N(B_2) = 1 + 1 = 2$, et $N(F,F_1,F_2) = 2 \times 1 = 2$. Il y a donc bien 2 graphes de connexions réalisables à partir de la fonction présentée sur la figure 15.

C-4-2) Exemple 2

Supposons maintenant que nous ayons à traiter la fonction suivante :



(Figure 35) Fonction F''

La matrice de connexions correspondante s'écrit :

	f	g	h	i	j	k
a	1	0	0	0	0	0
b	0	1	1	1	1	0
c	0	1	1	1	1	0
d	0	0	0	1	1	0
e	1	0	0	0	0	1

(Figure 36) Matrice de connexions de la fonction F''

soit, après permutations :

	i	j	g	h	f	k
b	1	1	1	1	0	0
c	1	1	1	1	0	0
d	1	1	0	0	0	0
e	0	0	0	0	1	1
a	0	0	0	0	1	0

(Figure 37) Matrice réordonnée

On obtient donc deux blocs :

	i	j	g	h
b	1	1	1	1
c	1	1	1	1
d	1	1	0	0

et

	f	k
e	1	1
a	1	0

(Figure 38) **Bloc B''_1** **Bloc B''_2**

Calcul de $N(B''_1)$

Nous cherchons ici $N(2,2,2,1)$, avec $n = 1$, $C = 4$ et $P = 3$. j peut prendre les valeurs $j = 1$ et $j = 2$ (au plus deux consommateurs connectées au producteur d).

* Pour $j = \gamma = 1$, le bloc équivalent restant est alors (3,2) (figure 39).

	i	g	h
b	1	1	1
c	1	1	1

ou

	j	g	h
b	1	1	1
c	1	1	1

(Figure 39) Blocs équivalents restants pour $j = \gamma = 1$.

Nous avons ici deux blocs équivalents restants, car on a $C_{c_0}^{\gamma}$ choix possibles pour la sélection d'un consommateur libre à connecter à d.

Pour chacun de ces blocs équivalents restants, on a $N(3,2) = 6$ graphes possibles, soit 12 graphes en tout.

* Pour $j = \gamma = 2$, le bloc équivalent restant est alors (2,2), et $N(2,2) = 2$.

	g	h
b	1	1
c	1	1

On a donc $N(B''_1) = 12 + 2 = 14$.

Calcul de $N(B''_2)$

D'après le calcul ci-dessus, on a $N(1,1,1,1) = 1$. Donc $N(B''_2) = 1$, et par suite $N(F'',F''_1,F''_2,F''_3) = 14 \times 1 = 14$.

On peut donc grâce à cet algorithme calculer le nombre de graphes de connexions possibles pour n'importe quelle fonction SAGA F pourvu que l'on connaisse son interface ainsi que celles de fonctions $F_1,\ldots, F_n$ qui la composent.

D) Mesure de complexité proposée

Soit $N_G(F, F_1,..., F_n)$ le nombre de graphes de connexions possibles pour une fonction SAGA F, décomposée en n sous-fonctions $F_1,..., F_n$. Avec les notations du chapitre 3, nous définissons alors la complexité d'assemblage R de la façon suivante :

$$R(F, F_1,..., F_n) = Log_2(N_G(F, F_1,..., F_n)).$$

$R(F, F_1,..., F_n)$ représente alors une mesure de la complexité relationnelle de F, et s'apparente au nombre de décisions binaires nécessaires pour sélectionner un graphe de connexions parmi tous les graphes de connexions possibles pour la fonction F, par analogie avec la théorie de l'information. Avec la formule du chapitre 3, nous obtenons la formule suivante pour la définition de la complexité globale d'une fonction SAGA :

$$C(F) = \sum_{\sigma \in B(F)} C(\sigma) + \sum_{\sigma \in D(F) \setminus B(F)} R(\sigma, S(\sigma)) \text{ , ce qui s'écrit}$$

$$C(F) = \sum_{\sigma \in B(F)} C(\sigma) + \sum_{\sigma \in D(F) \setminus B(F)} Log_2 [N_G(\sigma, S(\sigma))] \text{ , ou encore}$$

$$C(F) = \sum_{\sigma \in B(F)} C(\sigma) + Log_2 [\prod_{\sigma \in D(F) \setminus B(F)} N_G(\sigma, S(\sigma))].$$

Ce résultat fait apparaître le produit des nombres de graphes possibles pour chaque étape de la décomposition de la fonction F, ce qui correspond à considérer comme complexité globale d'une fonction la somme des complexités de ses fonctions élémentaires, plus le logarithme du nombre total de graphes de connexions possibles dans sa décomposition globale.

Pour des raisons exposées au début du chapitre 3, nous nous attachons ici essentiellement à étudier la complexité d'assemblage des fonctions décomposables, ce qui revient à supprimer le terme $\sum_{\sigma \in B(F)} C(\sigma)$ dans l'expression ci-dessus. On obtient alors

$$C(F) = Log_2 [\prod_{\sigma \in D(F) \setminus B(F)} N_G(\sigma, S(\sigma))].$$

La complexité d'une fonction SAGA s'exprime ainsi sous forme de complexité d'assemblage des fonctions de base de son arbre de décomposition. Deux fonctions pourront donc utiliser les mêmes fonctions élémentaires, et avoir néanmoins des complexités tout-à-fait différentes du fait de leurs décompositions différentes. De même, deux programmeurs réalisant la même fonction avec les mêmes fonctions de base pourront obtenir des résultats différents, autant du point de vue de la complexité des solutions retenues qu'en terme de présentation des solutions (placement des sous-fonctions et routage des connexions).

Dans le paragraphe suivant, nous interprétons cette remarque en considérant sur un exemple réel deux décompositions possibles, et en étudiant la complexité des deux solutions envisagées.

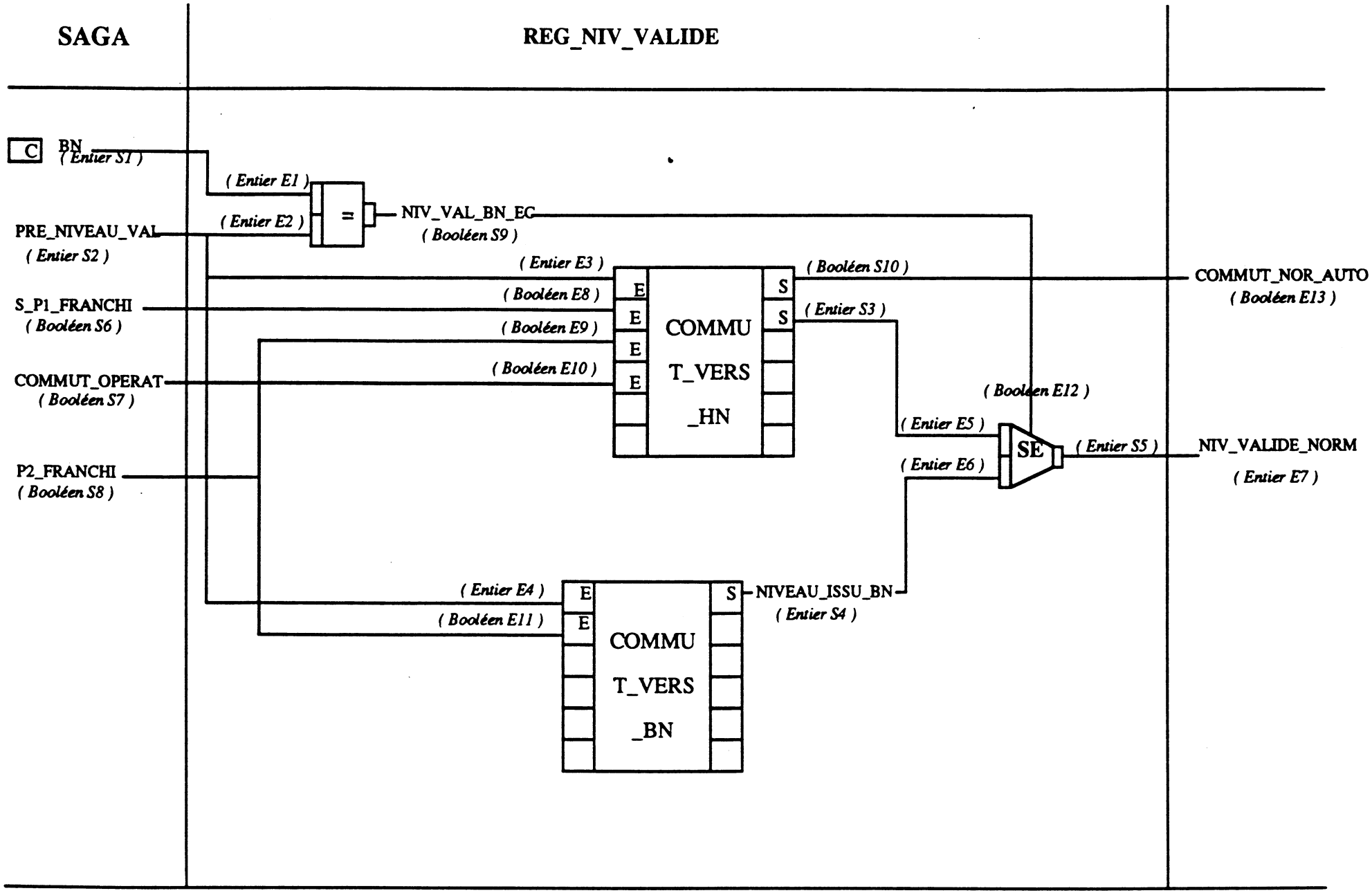
E) Exemple réel

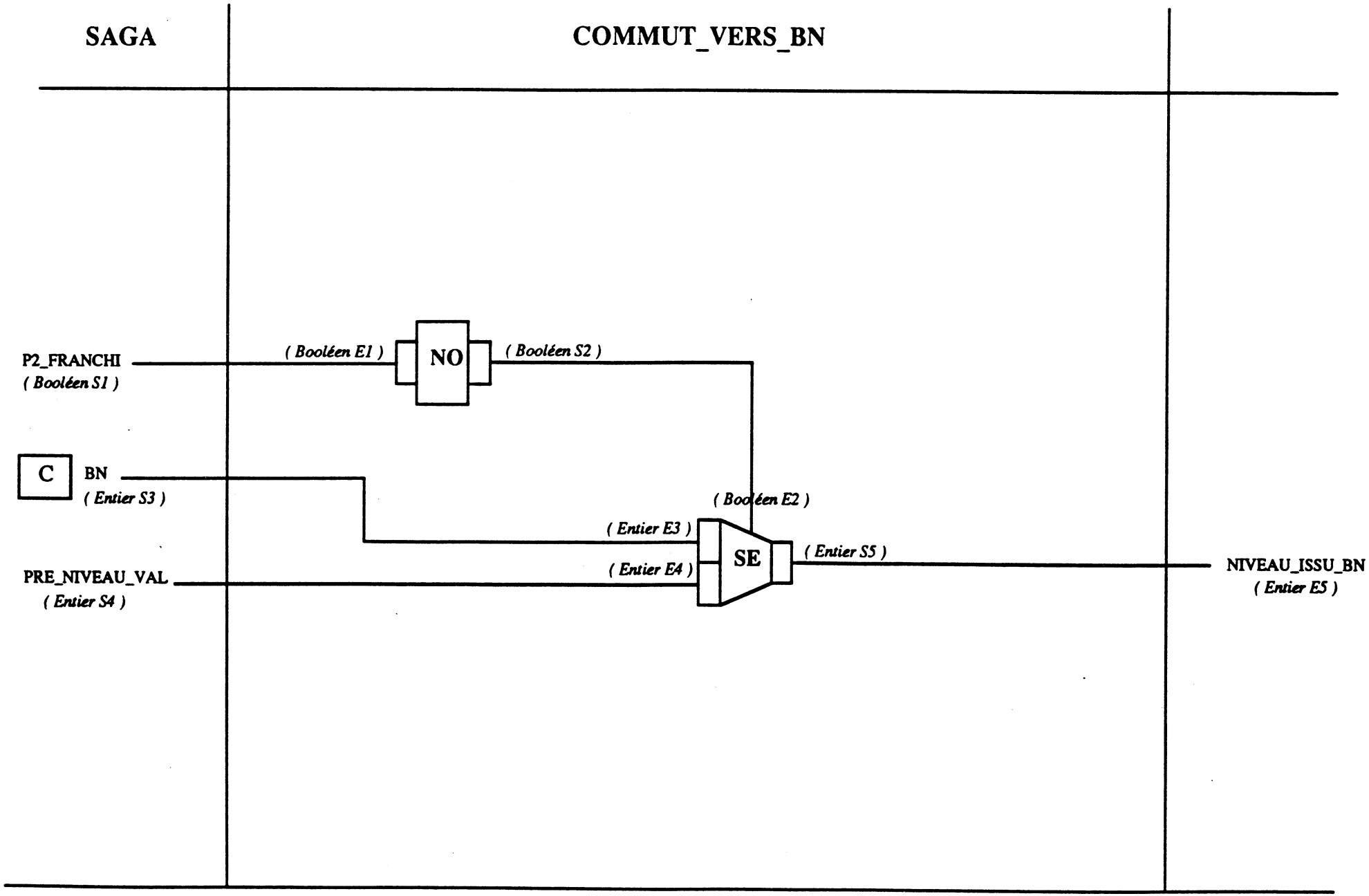
L'idée développée dans ce paragraphe est la suivante : une mesure de complexité définie sur des fonctions SAGA devrait permettre d'optimiser la décomposition de ces fonctions.

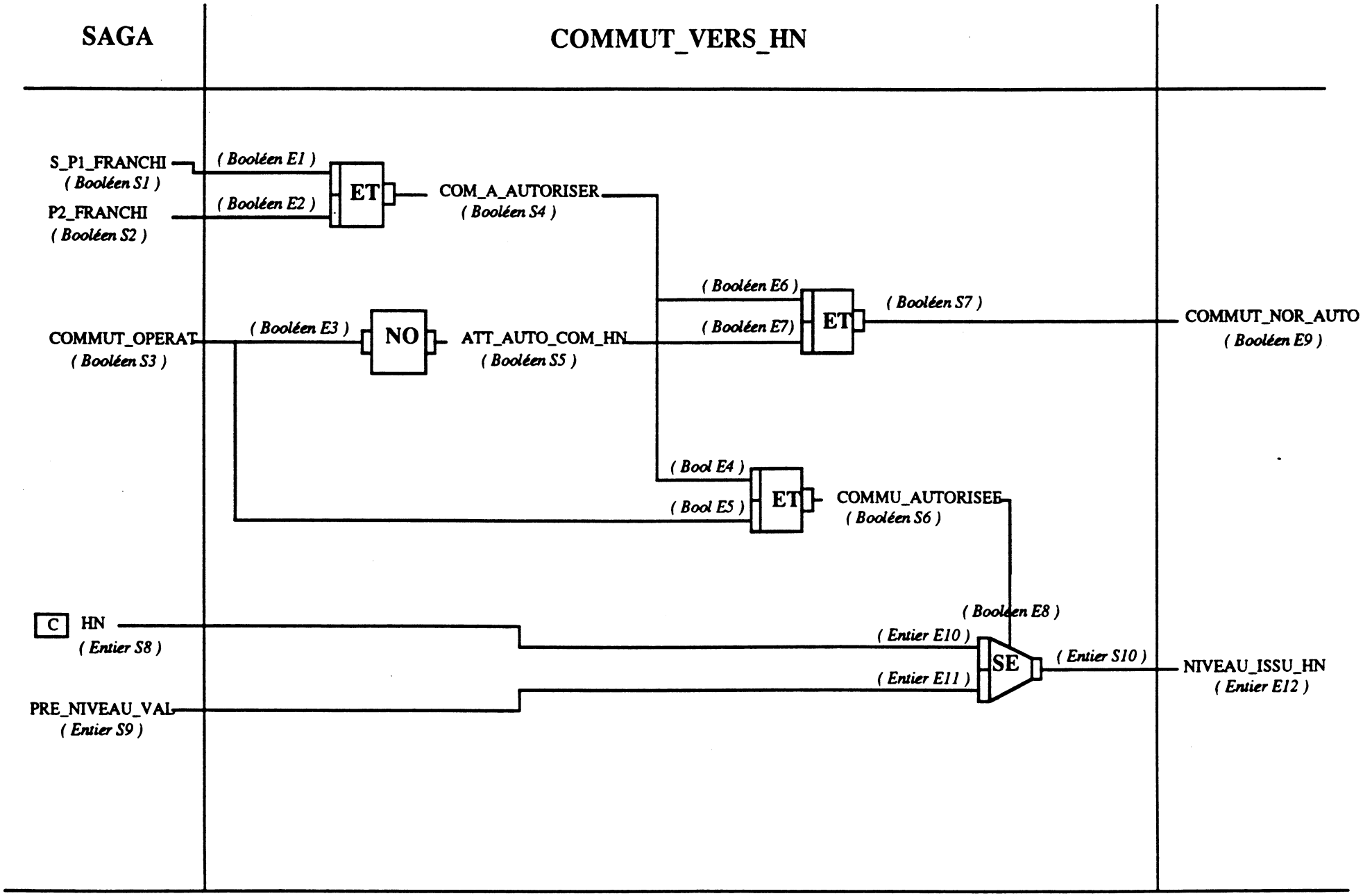
En effet, si on suppose donnés les composants d'une fonction (sous-fonctions et opérateurs SAGA), il existe plusieurs manières de décomposer la fonction, c'est-à-dire plusieurs façons de répartir les différentes tâches entre les sous-fonctions. Comme il existe nécessairement un nombre fini de répartitions possibles, on peut calculer celle dont la complexité est minimale, et ainsi proposer une démarche d'optimisation du développement des fonctions en cours de conception.

Pour illustrer ce propos, nous prenons l'exemple suivant (tiré d'un projet en cours à Merlin Gerin) : nous considérons la fonction REG_NIV_VALIDE, décomposée en deux sous-fonctions COMMUT_VERS_HN et COMMUT_VERS_BN. Les vues SAGA correspondantes sont présentées ci-après.

Les constantes utilisées dans la définition d'une fonction sont désignées dans la marge gauche de cette fonction par la lettre C. Ce sont les seules variables d'une fonction qui ne proviennent pas de la fonction du niveau supérieur.


SAGA
REG_NIV_VALIDE
C
BN
(Entier S1)
(Entier E1)
(Entier E2)
=
NIV_VAL_BN_EG
(Booléen S9)
PRE_NIVEAU_VAL
(Entier S2)
(Entier E3)
(Booléen E8)
(Booléen E9)
(Booléen E10)
E
E
E
E
COMMU
T_VERS
_HN
S
S
(Booléen S10)
(Entier S3)
COMMUT_NOR_AUTO
(Booléen E13)
S_P1_FRANCHI
(Booléen S6)
COMMUT_OPERAT
(Booléen S7)
(Booléen E12)
(Entier E5)
(Entier E6)
SE
(Entier S5)
NIV_VALIDE_NORM
(Entier E7)
P2_FRANCHI
(Booléen S8)
(Entier E4)
(Booléen E11)
E
E
COMMU
T_VERS
_BN
S
NIVEAU_ISSU_BN
(Entier S4)


SAGA
COMMUT_VERS_BN
P2_FRANCHI
(Booléen S1)
(Booléen E1)
NO
(Booléen S2)
C
BN
(Entier S3)
(Booléen E2)
(Entier E3)
(Entier E4)
SE
(Entier S5)
PRE_NIVEAU_VAL
(Entier S4)
NIVEAU_ISSU_BN
(Entier E5)


SAGA
COMMUT_VERS_HN
S_P1_FRANCHI
(Booléen S1)
P2_FRANCHI
(Booléen S2)
(Booléen E1)
(Booléen E2)
ET
COM_A_AUTORISER
(Booléen S4)
COMMUT_OPERAT
(Booléen S3)
(Booléen E3)
NO
ATT_AUTO_COM_HN
(Booléen S5)
(Booléen E6)
(Booléen E7)
ET
(Booléen S7)
COMMUT_NOR_AUTO
(Booléen E9)
(Bool E4)
(Bool E5)
ET
COMMU_AUTORISEE
(Booléen S6)
C
HN
(Entier S8)
PRE_NIVEAU_VAL
(Entier S9)
(Booléen E8)
(Entier E10)
(Entier E11)
SE
(Entier S10)
NIVEAU_ISSU_HN
(Entier E12)

Dans ces schémas, toutes les variables ont été numérotées : producteurs (variables S_n) ou consommateurs (variables E_n). Nous avons aussi fait figurer les types de ces variables, ici booléen (B, Bool) ou entier (Ent), de sorte qu'il est à présent facile de déterminer les matrices de connexions associées à ces fonctions :

	E_1	E_2	E_3	E_4	E_5
S_1	1	1	0	0	0
S_2	0	1	0	0	0
S_3	0	0	1	1	0
S_4	0	0	1	1	0
S_5	0	0	0	0	1

(Figure 40) Matrice de COMMUT_VERS_BN

	E_1	E_2	E_3	E_4	E_5	E_6	E_7	E_8	E_9	E_{10}	E_{11}	E_{12}
S_1	1	1	1	1	1	1	1	1	0	0	0	0
S_2	1	1	1	1	1	1	1	1	0	0	0	0
S_3	1	1	1	1	1	1	1	1	0	0	0	0
S_4	0	0	1	1	1	1	1	1	1	0	0	0
S_5	1	1	0	1	1	1	1	1	1	0	0	0
S_6	1	1	1	0	0	1	1	1	1	0	0	0
S_7	1	1	1	1	1	0	0	1	1	0	0	0
S_8	0	0	0	0	0	0	0	0	0	1	1	0
S_9	0	0	0	0	0	0	0	0	0	1	1	0
S_{10}	0	0	0	0	0	0	0	0	0	0	0	1

(Figure 41) Matrice de COMMUT_VERS_HN

	E_1	E_2	E_3	E_4	E_5	E_6	E_7	E_8	E_9	E_{10}	E_{11}	E_{12}	E_{13}
S_1	1	1	1	1	1	1	0	0	0	0	0	0	0
S_2	1	1	1	1	1	1	0	0	0	0	0	0	0
S_3	1	1	0	1	1	1	1	0	0	0	0	0	0
S_4	1	1	1	0	1	1	1	0	0	0	0	0	0
S_5	1	1	1	1	0	0	1	0	0	0	0	0	0
S_6	0	0	0	0	0	0	0	1	1	1	1	1	0
S_7	0	0	0	0	0	0	0	1	1	1	1	1	0
S_8	0	0	0	0	0	0	0	1	1	1	1	1	0
S_9	0	0	0	0	0	0	0	1	1	1	1	1	1
S_{10}	0	0	0	0	0	0	0	0	0	0	1	1	1

(Figure 42) Matrice de REG_NIV_VALIDE

Si on calcule la complexité d'assemblage associée à chacune de ces vues, on trouve :

- R (REG_NIV_VALIDE, COMMUT_VERS_HN, COMMUT_VERS_BN)
  $= \text{Log}_2 (N_G (\text{REG\_NIV\_VALIDE})) = 17.02$;
- R (COMMUT_VERS_HN) $= \text{Log}_2 (N_G (\text{COMMUT\_VERS\_HN})) = 19.89$;
- R (COMMUT_VERS_BN) $= \text{Log}_2 (N_G (\text{COMMUT\_VERS\_BN})) = 1$,

soit une complexité totale pour REG_NIV_VALIDE de

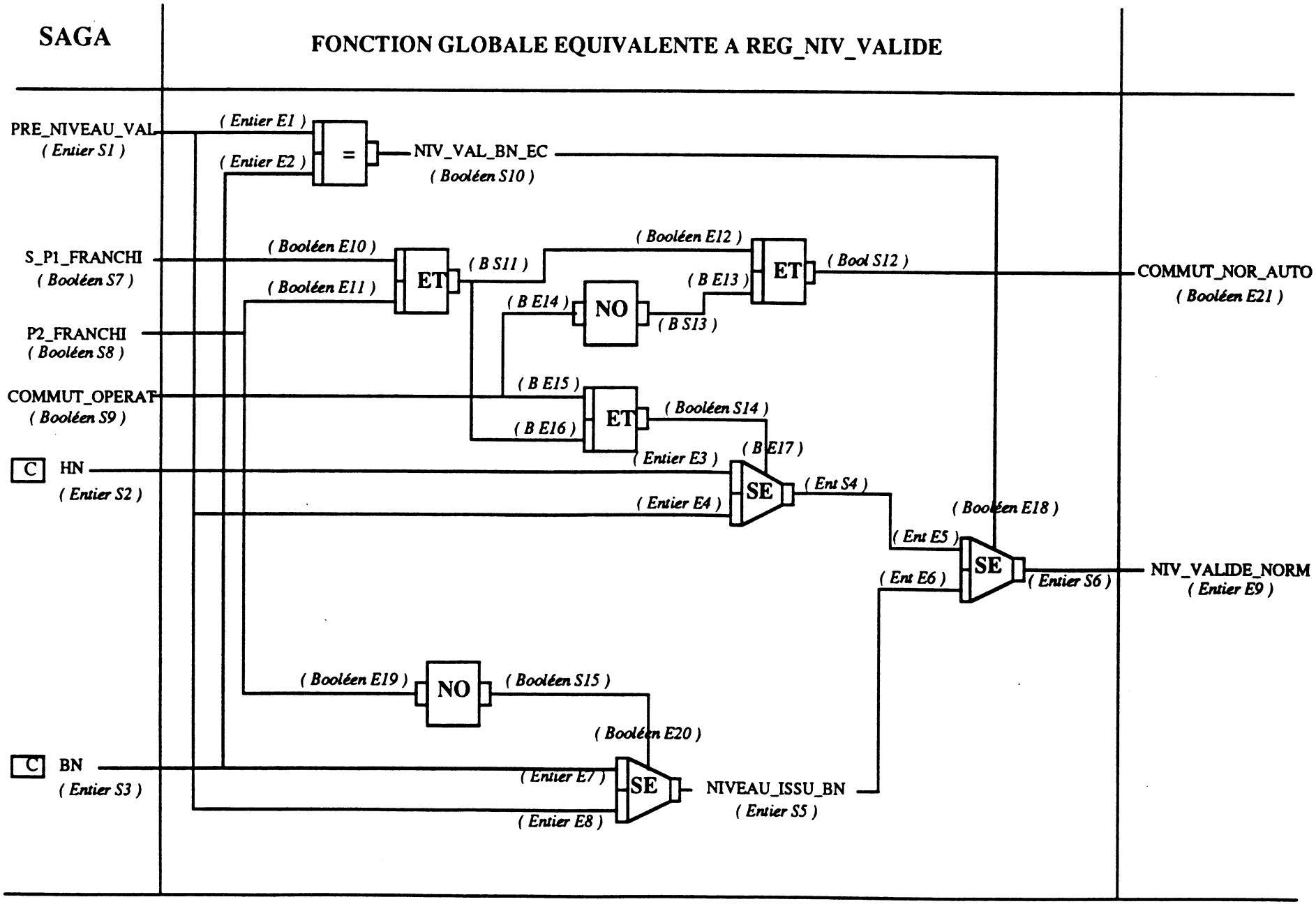
C (REG_NIV_VALIDE) = C (COMMUT_VERS_HN) + C (COMMUT_VERS_BN)
$+ \text{Log}_2 (N_G (\text{REG\_NIV\_VALIDE}))$, avec

C (COMMUT_VERS_HN) = R (COMMUT_VERS_HN), et

C (COMMUT_VERS_BN) = R (COMMUT_VERS_BN).

Nous avons donc C (REG_NIV_VALIDE) = 17.02 + 19.89 + 1 = 37.91 . C'est la complexité globale de la fonction REG_NIV_VALIDE.

Considérons à présent la fonction équivalente à REG_NIV_VALIDE, où l'on aurait fait figurer directement sur la vue correspondante toutes les fonctionnalités de la fonction ramenées au même niveau. Cette vue aurait l'allure suivante :


SAGA
FONCTION GLOBALE EQUIVALENTE A REG_NIV_VALIDE
PRE_NIVEAU_VAL
(Entier S1)
(Entier E1)
(Entier E2)
=
NIV_VAL_BN_EC
(Booléen S10)
S_P1_FRANCHI
(Booléen S7)
(Booléen E10)
(Booléen E11)
ET
(B S11)
(Booléen E12)
ET
(Bool S12)
COMMUT_NOR_AUTO
(Booléen E21)
(B E13)
(B E14)
NO
(B S13)
P2_FRANCHI
(Booléen S8)
COMMUT_OPERAT
(Booléen S9)
(B E15)
ET
(Booléen S14)
(B E16)
(B E17)
C
HN
(Entier S2)
(Entier E3)
SE
(Ent S4)
(Entier E4)
(Booléen E18)
(Ent E5)
SE
NIV_VALIDE_NORM
(Entier E9)
(Ent E6)
(Entier S6)
(Booléen E19)
NO
(Booléen S15)
(Booléen E20)
C
BN
(Entier S3)
(Entier E7)
SE
NIVEAU_ISSU_BN
(Entier S5)
(Entier E8)

Nous pouvons construire la matrice de connexions de cette fonction équivalente :

	E_1	E_2	E_3	E_4	E_5	E_6	E_7	E_8	E_9	E_{10}	E_{11}	E_{12}	E_{13}	E_{14}	E_{15}	E_{16}	E_{17}	E_{18}	E_{19}	E_{20}	E_{21}
S_1	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0
S_2	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0
S_3	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0
S_4	1	1	0	0	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0
S_5	1	1	1	1	1	1	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0
S_6	1	1	1	1	0	0	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0
S_7	0	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1	1	1	1	0
S_8	0	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1	1	1	1	0
S_9	0	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1	1	1	1	0
S_{10}	0	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1	1	1	1	1
S_{11}	0	0	0	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1	1	1
S_{12}	0	0	0	0	0	0	0	0	0	1	1	0	0	1	1	1	1	1	1	1	1
S_{13}	0	0	0	0	0	0	0	0	0	1	1	1	1	0	1	1	1	1	1	1	1
S_{14}	0	0	0	0	0	0	0	0	0	1	1	1	1	1	0	0	1	1	1	1	1
S_{15}	0	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1	1	0	1	1

(Figure 43) Matrice de la fonction équivalente

Puis on peut calculer la complexité de cette fonction. On trouve :

C (FONCTION EQUIVALENTE) = Log_2 (N_G (FONCTION EQUIVALENTE)) = 48.62 .

Cette dernière fonction apparaît plus complexe que la réunion des fonctions COMMUT_VERS_BN et COMMUT_VERS_HN pour former REG_NIV_VALIDE, car en plaçant sur un même plan toutes les variables disponibles dans ces trois fonctions, on autorise des connexions supplémentaires entre ces variables, ce qui n'était pas permis dans la configuration initiale. On augmente donc le nombre total de graphes de connexions possibles, et par là même, la complexité globale de la fonction.

L'évolution des langages de programmation est toujours allée dans le sens d'une plus grande structuration des programmes. Si nous prenons comme hypothèse qu'un programme est d'autant plus compréhensible qu'il est mieux découpé et structuré, alors nous constatons sur cet exemple que la mesure de complexité proposée satisfait cette intuition : le résultat fourni est en effet meilleur dans le cas où les fonctionnalités du programme ont été séparées que dans le cas où toutes les opérations sont effectuées à un même niveau.

Dans le cas présenté, la démarche consistant à scinder un problème principal en un certain nombre de sous-problèmes plus simples donne un meilleur résultat vis-à-vis de la mesure proposée que l'appréhension directe du problème initial dans sa globalité.

La mesure de complexité choisie privilégie donc la bonne structuration des programmes.

Il apparaît clairement selon ce raisonnement que l'on peut définir pour chaque fonction une répartition optimale de ses fonctionnalités par sous-fonctions. En effet, la fonction REG_NIV_VALIDE présentée ci-dessus utilise globalement (y compris dans ses sous-fonctions) :

- 1 opérateur "=" (test d'égalité) ;
- 3 opérateurs "ET" (multiplication booléenne) ;
- 3 opérateurs "NO" (complémentarité logique) ;
- 3 opérateurs "SE" (sélecteur [if...then...else]) , soit un total de 10 opérateurs.

Si on décide de doter REG_NIV_VALIDE de 2 sous-fonctions, alors il existe un certain nombre de combinaisons possibles pour la répartition de ces opérateurs parmi les sous-fonctions et la fonction principale. Dans l'exemple choisi, REG_NIV_VALIDE a conservé 3 opérateurs ("=", "NO", "SE"), et les sous-fonctions se sont partagées les opérateurs restants, soient 2 opérateurs pour COMMUT_VERS_BN ("NO" et "SE") et 5 opérateurs pour COMMUT_VERS_HN ("ET", "NO" et "SE").

On peut théoriquement calculer la répartition optimale de ces opérateurs, qui minimise la complexité globale de la fonction REG_NIV_VALIDE, comparer le comportement des fonctions correspondantes avec celui des fonctions initiales (retours d'expériences, erreurs en test,...), et décider du bien-fondé de la métrique en fonction de ces observations. Néanmoins, ce genre de calcul s'avère très long et fastidieux, car le processus de construction des sous-fonctions est difficilement automatisable (croissance exponentielle des possibilités en fonction du nombre d'opérateurs, constantes communes à plusieurs sous-fonctions, prise en compte des variables d'activation,...).

De plus, pour rester facilement compréhensible, chacune des sous-fonctions doit effectuer un ensemble de tâches cohérentes entre elles, et non pas une collection hétéroclite de bribes de tâches. En effet, en supposant que l'on ait pu exhiber toutes les configurations possibles à partir d'un ensemble d'opérateurs et comparer la complexité de chacune de ces configurations, il se peut que la configuration dont la complexité est minimale soit incompréhensible au niveau de sa fonction. Il nous semble en effet que la compréhension d'une fonction est directement liée au découpage de son graphe "flots de données" en sous-graphes de telle façon que ces sous-graphes représentent un calcul pour le concepteur.

La notion d'optimisation présente donc d'indéniables attraits d'un point de vue théorique, mais elle nous semble difficile à mettre en pratique. Peut-être faudrait-il penser à utiliser les techniques mises en œuvre dans les optimiseurs de code.

F) Cas d'un grand nombre de variables - Approximation

Nous avons vu dans le paragraphe C que l'on pouvait calculer le nombre de graphes associé à n'importe quelle fonction SAGA.

Mais la formule trouvée pour le calcul du nombre de graphes associés à un bloc (cf. § C-3-3) est récursive. Or nous nous sommes rapidement rendus compte que le temps de calcul de ce nombre devenait rédhibitoire pour $C \geq 25$ (plus d'une demi-journée sur un mini-ordinateur...). Il nous a donc fallu déterminer une approximation judicieuse de la complexité en fonction des caractéristiques de la matrice de connexions.

Au vu du tableau de la figure 27, nous pouvons sans craindre de nous tromper beaucoup, estimer que le nombre de graphes est une fonction quasi-exponentielle du nombre de producteurs P ou du nombre de consommateurs C. Nous pouvons donc supposer que le logarithme de ce nombre de graphes est linéaire par rapport à ces caractéristiques de la matrice. C'est l'hypothèse que nous avons faite.

Nous avons alors cherché à estimer le logarithme du nombre de graphes associé à un bloc par une régression linéaire multiple sur certaines caractéristiques du bloc. Dans un premier temps, nous avons choisi 4 grandeurs décrivant le bloc. Ces grandeurs sont les suivantes :

- n (nombre de familles du bloc) ;
- $C = \sum_{k=0}^{n} c_k$ (nombre de consommateurs) ;
- $P = \sum_{k=0}^{n} p_k$ (nombre de producteurs) ;
- $n_0 = \sum_{k=1}^{n} c_k * p_k$ (nombre de cases interdites du bloc) ;

Ces 4 grandeurs ne sont pas suffisantes pour définir complètement le bloc. En effet, le nombre de cases interdites d'un bloc peut se répartir sur tous les sous-blocs nuls, et donc de façon différente pour deux blocs ayant néanmoins des valeurs de n, P, C et n_0 égales. Ces grandeurs présentent toutefois l'avantage de définir chaque bloc par un même nombre de paramètres : si nous avions choisi de décrire complètement le bloc, il aurait été nécessaire de lui associer la taille de chacun de ses sous-blocs nuls, donc un nombre de paramètres différent pour chacun d'eux ; nous n'aurions pas pu faire un calcul de régression.

Nous avons calculé la complexité d'un certain nombre de blocs-test, et nous avons cherché à savoir comment nos résultats pouvaient être approchés par un modèle linéaire. Après

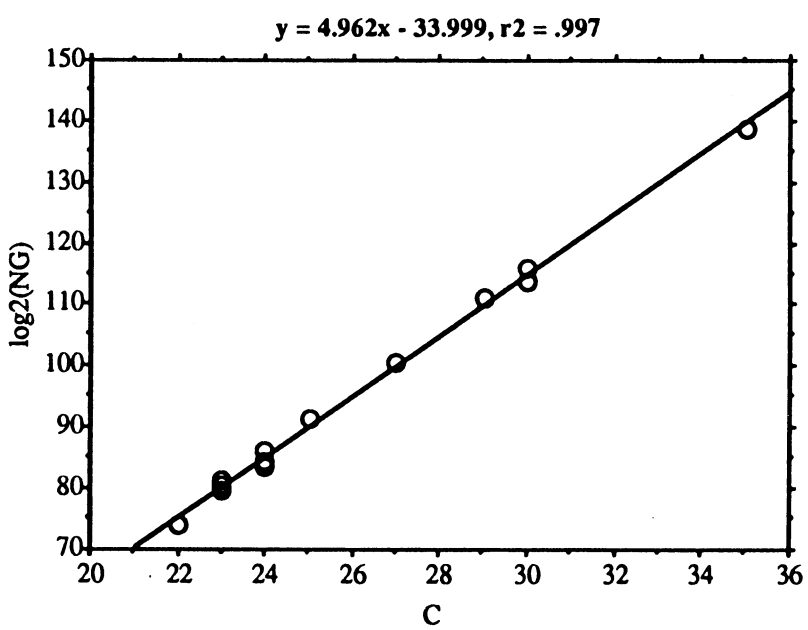
calculs sur 14 blocs-tests avec les 4 grandeurs retenues, nous obtenons la régression suivante :

$$Log_2(NG(B)) = -38.23 - 0.21\ P + 5.12\ C + 0.39\ n - 0.02\ n_0\ .$$

Nous constatons que la grandeur C est la seule qui entre significativement en considération dans la régression. Si l'on cherche plus précisément à corréler la complexité du bloc B $Log_2(NG(B))$ avec chacune des 4 variables retenues, on obtient :

corrélation($Log_2(NG(B))$, P) = 0.862 ;
corrélation($Log_2(NG(B))$, C) = 0.998 ;
corrélation($Log_2(NG(B))$, n) = 0.726 ;
corrélation($Log_2(NG(B))$, n_0) = 0.739.

La variable C peut donc à elle seule expliquer la variation de $Log_2(NG(B))$. Si nous calculons la régression simple de $Log_2(NG(B))$ sur C, nous obtenons en effet la courbe suivante :



(Figure 44) Régression sur la complexité

Nous avons donc $Log_2(NG(B)) \approx 4.962\ C - 33.999$, ce qui donne $NG(B) \approx \frac{31.17^C}{2^{34}}$.
C'est un moyen extrêmement rapide d'estimer la complexité quand le calcul devient trop long.

Chapitre 4 ETUDE EMPIRIQUE

Pour étudier les propriétés de la métrique présentée dans ce document, nous avons mis sur pied une campagne de mesures sur les différentes applications SAGA développées chez Merlin Gerin. Dans une première partie, nous décrirons l'environnement de cette campagne de mesures, puis nous analysons les résultats obtenus.

A) Campagne de mesures

Comme nous l'avons souligné dans l'introduction, ce qui nous intéresse, c'est de pouvoir obtenir des critères de qualité sur les logiciels SAGA dès la phase de conception, et non pas seulement durant les phases de tests. La campagne de mesures pour notre métrique s'est donc déroulée durant la phase de conception des applications SAGA.

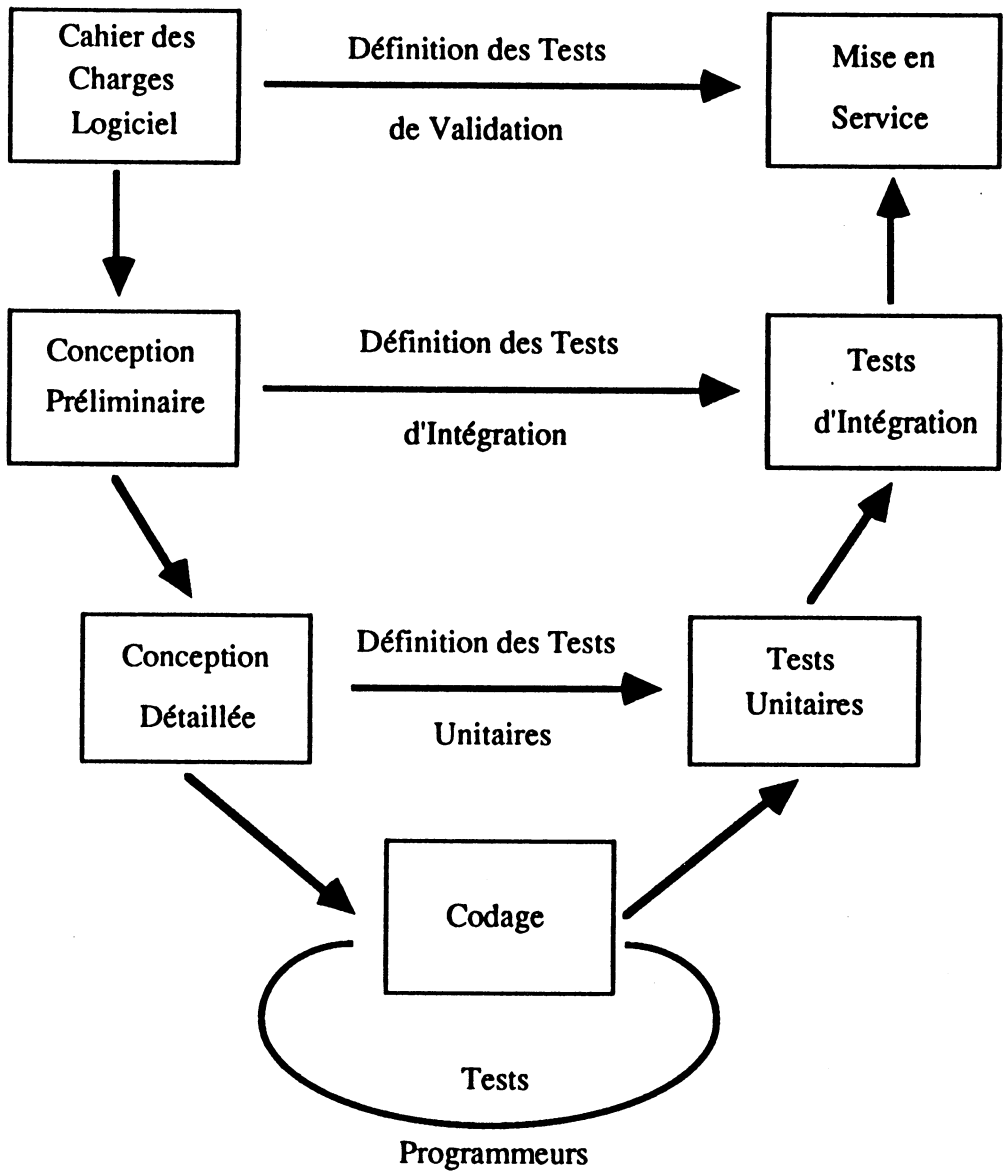
Il était exclu de mettre en œuvre une campagne de mesures pour corréler les valeurs de la complexité trouvées avec la propension des fonctions à contenir des fautes, puisque nous n'avons pas de relevé de fautes. Mais nous avons quand même essayé d'étudier la variation de la métrique proposée sur les différentes versions d'une même fonction.

Nous avons principalement voulu étudier les relations possibles entre la mesure de complexité proposée et d'autres mesures plus simples que l'on peut prendre sur les fonctions SAGA. C'est le résultat de cette démarche que nous présentons ici.

A-1) Cycle de vie des logiciels étudiés

Les logiciels conçus dans le département SES de Merlin Gerin bénéficient d'un cycle de vie différent suivant qu'il s'agit de logiciels critiques ou non (on qualifie de "critique" un logiciel dont la défaillance peut mettre en danger la vie d'êtres humains, par exemple le programme de contrôle du cœur d'un réacteur de centrale nucléaire). Dans le cas de logiciels critiques, chacune des étapes du cycle de vie est formalisée et fait l'objet d'un document de référence.

Le cycle de vie standard appliqué aux logiciels à Merlin Gerin est le suivant :



(Figure 45) Cycle de vie des logiciels étudiés

En particulier, on peut se représenter schématiquement le traitement des logiciels de sûreté comme confié à trois équipes distinctes et indépendantes selon l'état d'avancement du programme : l'équipe des programmeurs conçoit et réalise le logiciel, celle des vérificateurs le valide, enfin celle de maintenance intervient durant la vie opérationnelle de l'application.

Sur le schéma ci-dessus, nous pouvons distinguer 7 phases principales :

1) Cahier des Charges Logiciel : il est mis au point en collaboration avec le client. Pendant cette phase sont aussi définis les tests finals de validation de l'application ;

2) Conception Préliminaire : définition de l'architecture globale de l'application, division en fonctions et détermination des relations entre ces fonctions ; on définit également ici les tests d'intégration de l'application ;
3) Conception Détaillée : affinement des fonctions afin d'obtenir des composants logiciels relativement petits, avec des entrées-sorties parfaitement connues, et dont la fonction est complètement cernée. On définit ici également les tests unitaires (i.e. pour chaque fonction) de l'application ;
4) Codage : écriture des composants logiciels dans le langage choisi. Tests Programmeurs : tests effectués par le programmeur pour vérifier le code produit ;
5) Tests Unitaires : tests en aveugle ; le vérificateur effectue les tests mis au point lors de la Conception Détaillée, et prend des mesures statiques sur le code (mesures de Halstead, Mac Cabe, Mohanty, ...). L'acceptation de la fonction dépend des résultats de ces tests. La valeur des mesures par rapport à des standards de complexité cyclomatique (Mac Cabe) ou de volume de programme (Halstead), permet soit de rejeter le composant, soit de l'accepter, en prenant soin d'étoffer les tests ultérieurs des modules présentant une forte complexité ;
6) Tests d'Intégration : relecture critique des fonctions et passage des tests définis lors de la Conception Préliminaire ;
7) Tests de Validation : relecture critique des documents et passage des tests définis lors de la mise au point du Cahier des Charges Logiciel.

Dans notre cas, une partie de la phase 1 ainsi que l'ensemble des phases 2 et 3 sont effectuées sous l'environnement SAGA comme décrit dans le chapitre 2. Le codage de la phase 4 est partiellement automatisé, l'outil SAGA générant automatiquement le programme C équivalent à l'application (seules des fonctions bien définies sont programmées manuellement pour être intégrées à l'application). Les différentes phases de test sont alors effectuées sur les programmes C correspondants.

A-2) Aspect perturbateur

Pour ne pas perturber le travail du programmeur ni son environnement, nous avons choisi de mettre au point une campagne de mesures "transparente" vis-à-vis des personnes utilisant l'outil SAGA. Ainsi les programmeurs SAGA n'ont-ils pas remarqué de différences dans leur travail quotidien pendant le déroulement de la campagne de mesures.

Mais ces programmeurs savaient qu'une telle campagne était en cours sur l'ensemble des applications SAGA, et il est heureux que cette idée ait été bien acceptée par tous ; en effet, la nécessité d'une méthode de validation a priori des logiciels est une idée déjà claire pour tous les programmeurs travaillant sur un projet nucléaire : cet état d'esprit rend la tâche des vérificateurs plus aisée, et l'idée d'une campagne de mesures tout-à-fait supportable.

De façon à ne gêner aucunement les programmeurs SAGA, nous avons choisi de récolter au cours de la journée le strict minimum d'informations nécessaires, et de reporter les calculs au cours de la nuit.

A-3) Organisation, mise en œuvre

Chaque fois qu'un programmeur crée ou modifie une fonction SAGA, il est obligé de valider ces changements par une touche spéciale. C'est le moment qui nous a semblé opportun pour intervenir et garder une trace des modifications, en notant dans un fichier le nom de l'application et de la fonction modifiées. Au cours de la nuit, on effectue alors toutes les mesures voulues sur les fonctions repérées dans la journée, puis on stocke les résultats.

Chaque jour, une moyenne de 5 à 6 applications sont modifiées, chacune en moyenne sur 4 à 5 fonctions différentes, soit environ 25 modifications en tout, donc 25 nouveaux enregistrements dans les fichiers de résultats.

Nous pouvons noter ici que l'unité de temps choisie est le jour. En effet, si une même fonction a subi plusieurs modifications au cours de la même journée, on ne pourra recueillir d'informations que sur la version la plus récente. Dans la pratique, cette remarque n'a pas grande importance. Il arrive en effet que plusieurs modifications successives soient apportées à une même fonction au cours d'une journée ; mais toutes ces modifications, sauf la dernière, sont des modifications artificielles dues au manque de souplesse d'utilisation de la version courante de l'outil SAGA, et sont non significatives.

Le traitement des données d'une journée comprend dans l'ordre :
- l'extraction des informations concernant les fonctions modifiées ou créées ;
- le calcul des complexités associées aux matrices de connexions correspondantes ;
- le stockage des informations collectées, et les différents traitements statistiques.

Les mesures effectuées sur les fonctions modifiées durant la journée ont été de plusieurs types : tout d'abord le calcul de la complexité, puis la prise de mesures triviales, ensuite détermination de la profondeur à laquelle se situe la fonction dans l'arbre de décomposition de l'application, et enfin la date.

Les mesures triviales collectées sont :
- le nombre de sous-fonctions SAGA de la fonction courante ;
- le nombre d'opérateurs SAGA entrant dans la décomposition de la fonction ;
- le nombre d'entrées de la fonction ;
- le nombre de sorties de la fonction ;
- le nombre de constantes utilisées dans la fonction ;
- le nombre total de types utilisés pour les variables internes de la fonction ;
- le nombre total de variables accessibles dans la fonction.

A la fin de cette campagne, on est en mesure d'apprécier après chaque modification de fonction le nombre de modifications restant à faire sur cette fonction ; on inclut donc cette information dans le fichier final, et les enregistrements présentent alors l'allure suivante :

NM_F || N_S_F || N_OP || N_E || N_S || N_C || N_VA || N_T || CPXT || DATE || PROF || N_VE

où les différents symboles représentent :

NM_F	:	nom de la fonction ;
N_S_F	:	nombre de sous-fonctions ;
N_OP	:	nombre d'opérateurs ;
N_E	:	nombre d'entrées ;
N_S	:	nombre de sorties ;
N_C	:	nombre de constantes ;
N_VA	:	nombre total de variables ;
N_T	:	nombre de types différents ;
CPXT	:	complexité d'assemblage ;
DATE	:	date de la modification ;
PROF	:	profondeur de la fonction dans l'arbre de décomposition ;
N_VE	:	nombre de versions ultérieures de la même fonction dans l'application.

A-4) Résultats

La campagne de mesures a commencé au début du mois de mars 1989 pour se terminer à la mi-juillet. Nous pouvons noter ici que lors du démarrage de la campagne de mesures, le développement de certaines applications était déjà bien avancé. Nous avons donc défini comme premières versions celles que nous avions à notre disposition à ce moment. En tout, 59 applications SAGA ont fait l'objet de mesures de complexité. Sur ces 59 applications, environ 43 sont significatives (les autres sont des applications de démonstration, ou des applications de test).

Tous les résultats d'une application sont réunis dans un fichier propre à cette application, les différents enregistrements étant classés dans l'ordre chronologique.

A partir de ces 59 fichiers, nous avons constitué un nouveau fichier contenant l'ensemble des enregistrements des fichiers de résultats, sur lequel nous allons pouvoir étudier les liens entre les différentes métriques.

D'autre part, nous avons calculé et stocké chaque soir la complexité globale (définie dans le paragraphe D du chapitre 3) des applications significatives, afin de pouvoir étudier la progression de cette complexité globale au cours du temps.

B) Exploitation des résultats

Nous nous intéressons ici essentiellement à :
- l'étude statistique des différentes observations, avec détermination de quelques lois de probabilité associées aux variables aléatoires correspondant aux observations ;
- la corrélation entre ces observations ;
- l'étude de la progression de la métrique de complexité au cours des différentes versions d'une même fonction ;
- l'étude de la morphologie d'une application SAGA (ce point sera abordé dans les paragraphes B-2 et B-3) ;
- l'étude de la progression de la métrique de complexité d'une application au cours du temps (cf. § B-4).

B-1) Aide à la conception

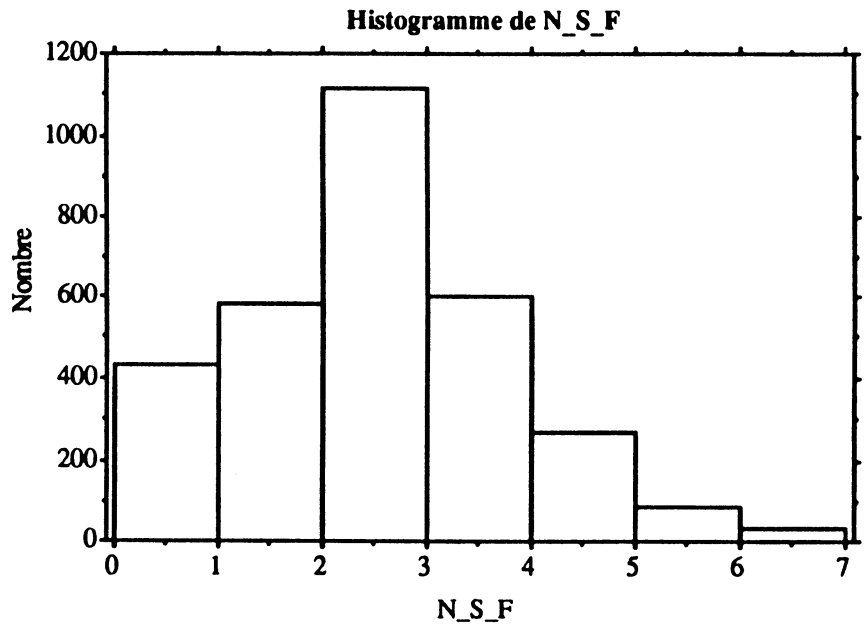
Dans ce paragraphe, nous traiterons des trois premiers points évoqués ci-dessus.

B-1-1) Etude statistique des observations

Nous pouvons tout d'abord analyser les résultats globaux sur l'ensemble des applications observées. Si l'on s'intéresse aux tris à plat des variables, on obtient les statistiques élémentaires et les histogrammes suivants :

1 . N_S_F : Nombre de sous-fonctions par fonction étudiée

Ce nombre varie entre 0 et 6, puisque l'outil SAGA limite à 6 le nombre de sous-fonctions par fonction. Nous présentons ci-dessous l'histogramme des données recueillies ainsi que les valeurs des statistiques élémentaires correspondantes.



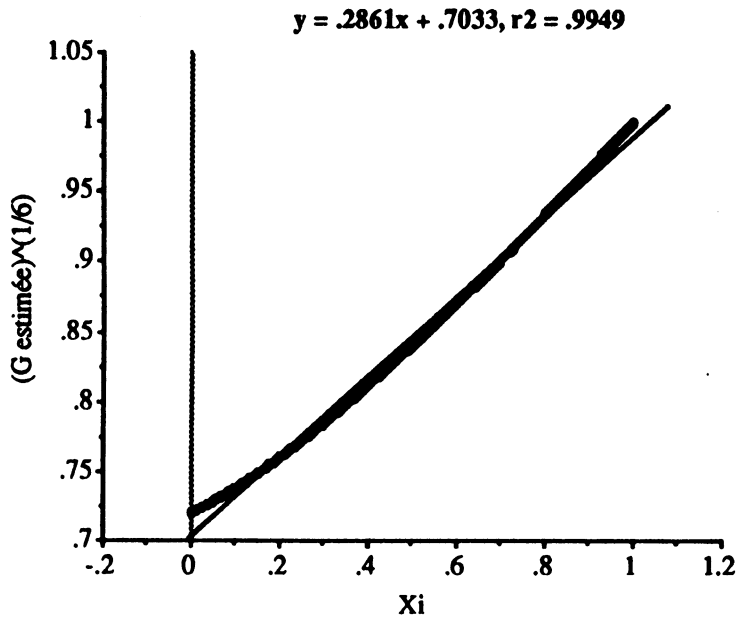
Mean:	Std. Dev.:	Std. Error:	Variance:	Coef. Var.:	Count:
2.032	1.311	.023	1.72	64.524	3126
Minimum:	**Maximum:**	**Range:**	**Sum:**	**Sum Squared:**	**# Missing:**
0	6	6	6353	18285	0

(Figure 46) Etude du nombre de sous-fonctions

On trouve ici l'espérance des observations (2 sous-fonctions par fonction en moyenne), l'écart-type, la variance , le nombre total d'observations (count), leur étendue (range), la somme des observations et celle de leurs carrés, enfin le nombre de données manquantes (# missing).

Si nous admettons que le nombre de sous-fonctions par fonction SAGA peut être considéré comme une variable aléatoire, alors le fait que cette quantité soit entière et ne puisse varier qu'entre 0 et 6, ainsi que l'allure de la courbe, nous suggère de tester l'adéquation à une loi binomiale.

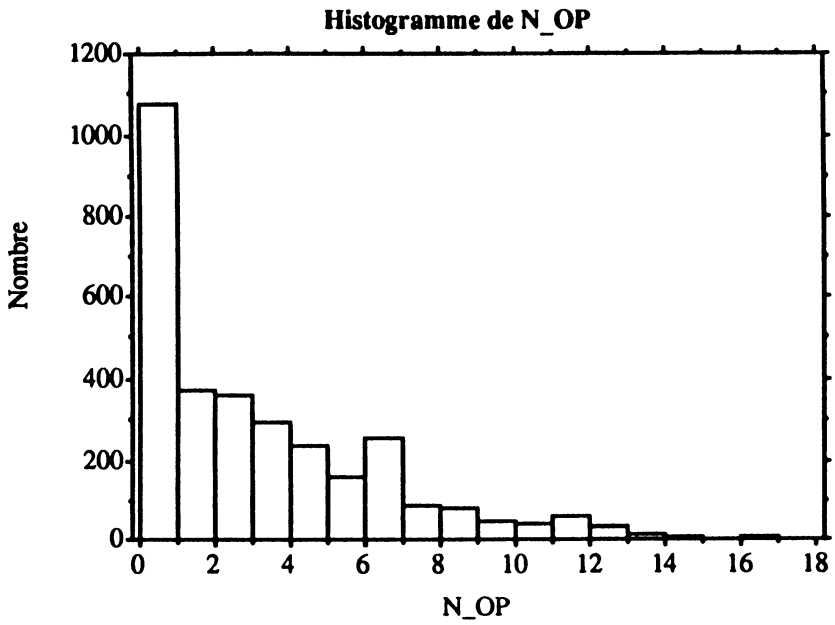
Pour ce faire, nous calculons la fonction génératrice empirique $\tilde{G}(s)$ grâce aux observations. Si notre hypothèse est correcte, alors la véritable fonction génératrice est $G(s) = (ps + q)^6$, où p est le paramètre de cette loi, et q = 1-p. La racine sixième de notre fonction génératrice empirique $\tilde{G}(s)$ devrait donc admettre une bonne régression linéaire ; si nous traçons la courbe $y = \sqrt[6]{\tilde{G}(x)}$ en faisant une régression linéaire simple, nous obtenons en effet :



(Figure 47) Loi du nombre de sous-fonctions

r^2 désigne ici le coefficient d'ajustement entre les observations et la régression linéaire associée (cette régression est d'autant meilleure que ce coefficient se rapproche de 1). L'adéquation est bonne ; on pourra donc admettre que le nombre de sous-fonctions par fonction suit bien une loi binomiale, et l'estimateur du paramètre p par cette méthode est $\tilde{p} = 0.29$. Dans le paragraphe B-3, nous utiliserons ce résultat pour modéliser la génération du graphe de décomposition d'une application SAGA.

2 . N_OP : Nombre d'opérateurs par fonction

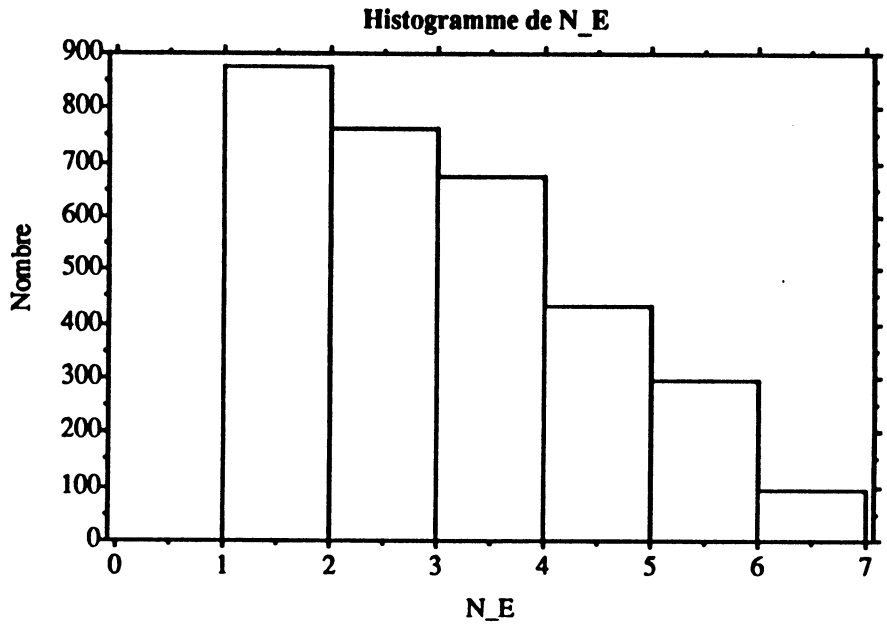


Mean:	Std. Dev.:	Std. Error:	Variance:	Coef. Var.:	Count:
2.787	3.188	.057	10.166	114.418	3126
Minimum:	**Maximum:**	**Range:**	**Sum:**	**Sum Squared:**	**# Missing:**
0	26	26	8711	56043	0

(Figure 48) Etude du nombre d'opérateurs

On peut noter ici que plus du tiers des fonctions étudiées ne possèdent aucun opérateur dans leur décomposition immédiate.

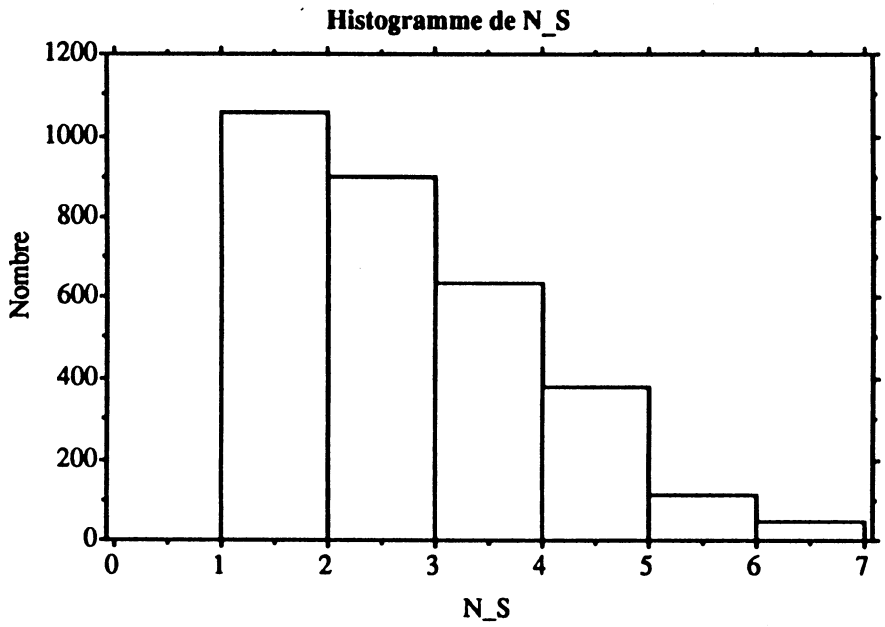
3 . N_E : Nombre d'entrées de la fonction



Mean:	Std. Dev.:	Std. Error:	Variance:	Coef. Var.:	Count:
2.611	1.411	.025	1.991	54.054	3126
Minimum:	**Maximum:**	**Range:**	**Sum:**	**Sum Squared:**	**# Missing:**
1	6	5	8161	27529	0

(Figure 49) Etude du nombre d'entrées

4 . N_S : Nombre de sorties de la fonction

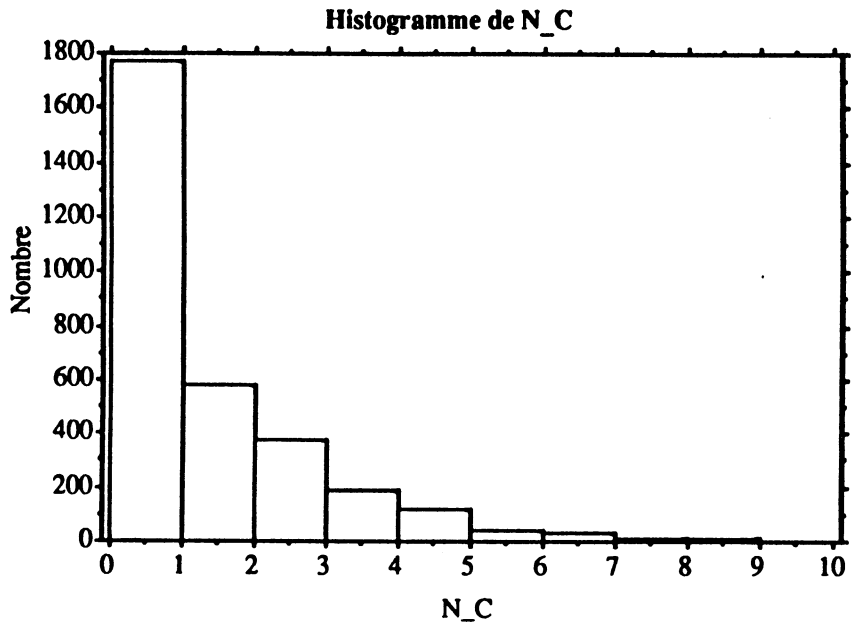


Mean:	Std. Dev.:	Std. Error:	Variance:	Coef. Var.:	Count:
2.274	1.226	.022	1.502	53.893	3126
Minimum:	**Maximum:**	**Range:**	**Sum:**	**Sum Squared:**	**# Missing:**
1	6	5	7109	20861	0

(Figure 50) Etude du nombre de sorties

Les répartitions du nombre d'entrées et du nombre de sorties d'une fonction SAGA sont similaires, mais il entre en moyenne plus de données dans une fonction SAGA qu'il n'en ressort.

5 . N_C : Nombre de constantes utilisées dans la fonction

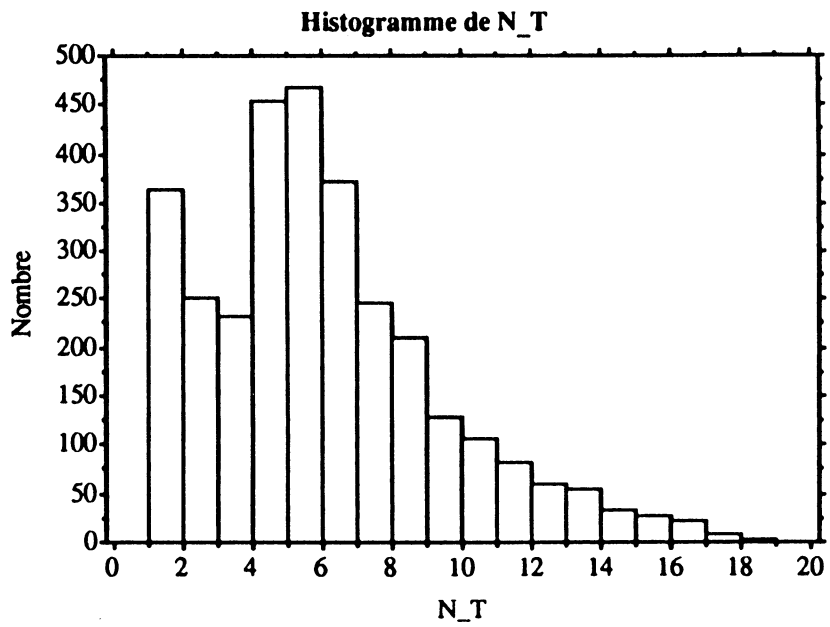


Mean:	Std. Dev.:	Std. Error:	Variance:	Coef. Var.:	Count:
.941	1.443	.026	2.081	153.24	3126
Minimum:	**Maximum:**	**Range:**	**Sum:**	**Sum Squared:**	**# Missing:**
0	11	11	2943	9275	0

(Figure 51) Etude du nombre de constantes

Plus de la moitié des fonctions mesurées n'utilisent aucune constante.

6 . N_T : Nombre de types différents représentés par les variables de la fonction

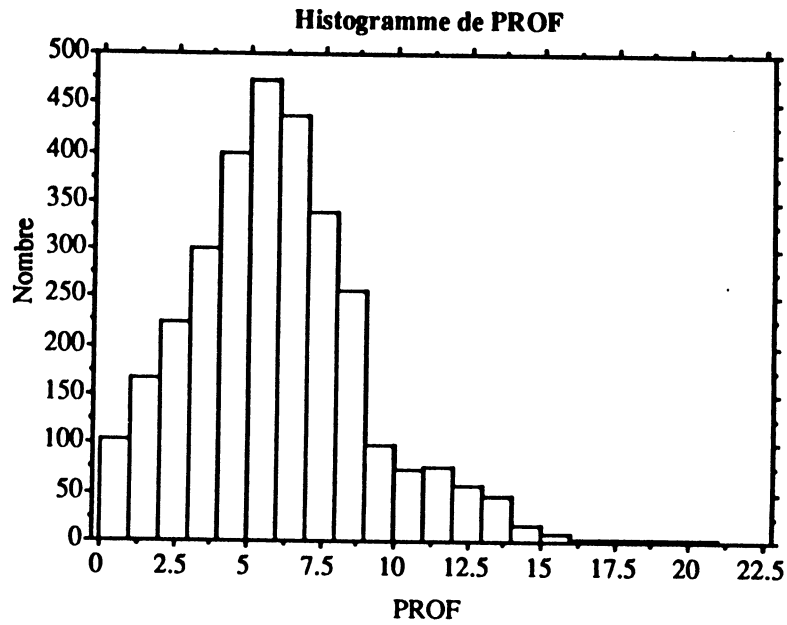


Mean:	Std. Dev.:	Std. Error:	Variance:	Coef. Var.:	Count:
5.585	3.481	.062	12.12	62.337	3126
Minimum:	**Maximum:**	**Range:**	**Sum:**	**Sum Squared:**	**# Missing:**
1	25	24	17458	135374	0

(Figure 52) Etude du nombre de types

La plupart des fonctions observées comportent entre 3 et 7 types différents selon leur taille.

7 . PROF : Profondeur de la fonction dans l'arbre de décomposition de l'application

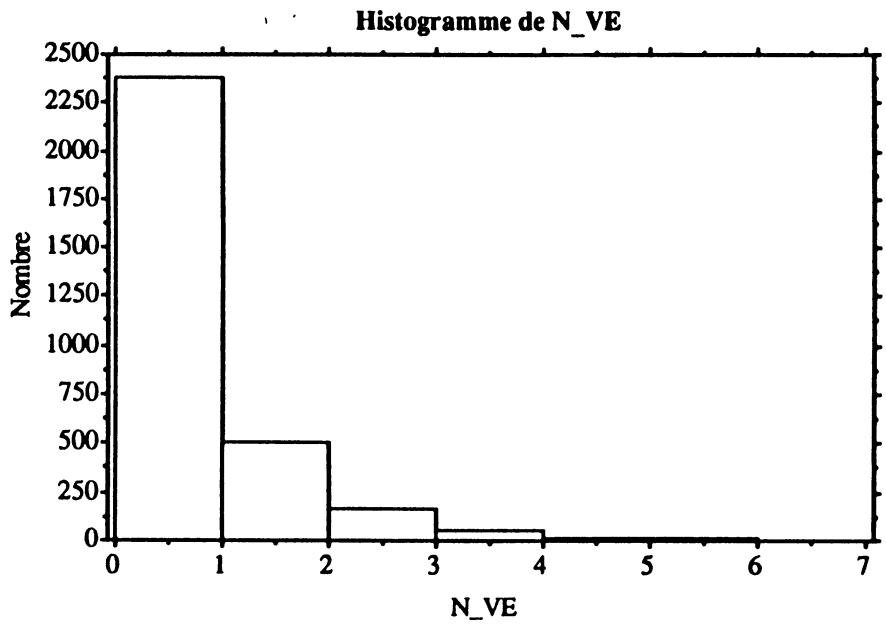


Mean:	Std. Dev.:	Std. Error:	Variance:	Coef. Var.:	Count:
5.467	3.068	.055	9.415	56.126	3093
Minimum:	**Maximum:**	**Range:**	**Sum:**	**Sum Squared:**	**# Missing:**
0	21	21	16909	121549	33

(Figure 53) Etude de la profondeur des fonctions

Nous pouvons noter ici que 33 enregistrements n'ont pas été pris en compte dans les calculs : il s'agit de fonctions intermédiaires créées, modifiées puis supprimées purement et simplement. Elles n'ont donc pas d'existence physique dans l'arbre de décomposition de l'application à la fin de la campagne de mesures, et donc pas de profondeur associée.

8 . N_VE : Nombre de versions nécessaires à l'obtention de la version définitive



(Figure 55) Histogramme du nombre de versions

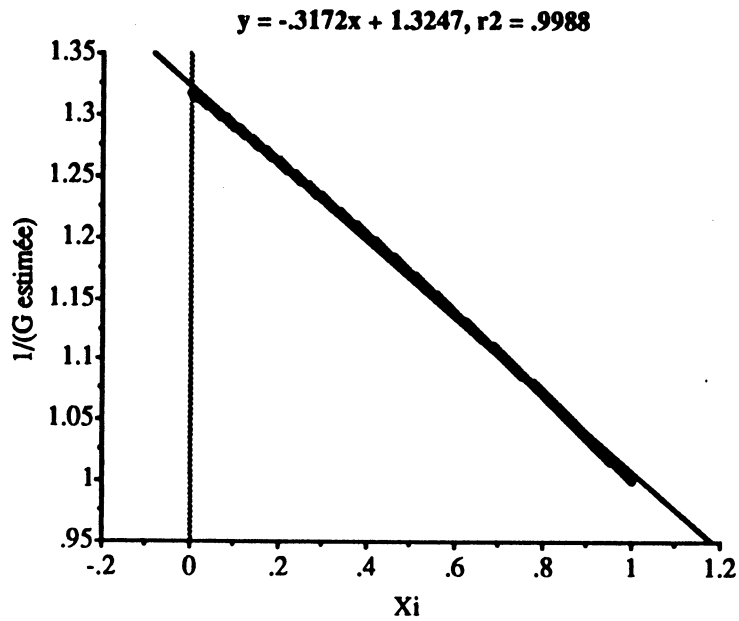
Nous constatons tout de suite sur ce diagramme que nous aurons des difficultés pour étudier la progression de la complexité d'une fonction au cours de ses versions successives, puisque 3 fonctions seulement ont été développées avec 6 versions successives, et que 3 fonctions sur 4 n'ont jamais été retouchées durant la campagne de mesures.

N_VE

Mean:	Std. Dev.:	Std. Error:	Variance:	Coef. Var.:	Count:
.364	.776	.014	.602	212.872	3126
Minimum:	**Maximum:**	**Range:**	**Sum:**	**Sum Squared:**	**# Missing:**
0	6	6	1139	2295	0

(Figure 56) Statistiques sur le nombre de versions

L'histogramme nous conduit à supposer que la loi de cette variable aléatoire est une loi géométrique décalée (dite encore loi de Pascal). La fonction génératrice d'une telle loi étant $G(s) = \frac{p}{1-qs}$ avec $q = 1 - p$, la courbe $y = \frac{1}{\widetilde{G}(s)}$ devrait pouvoir facilement être approchée par une régression linéaire. Et en effet, si l'on trace cette courbe, on obtient le diagramme suivant :

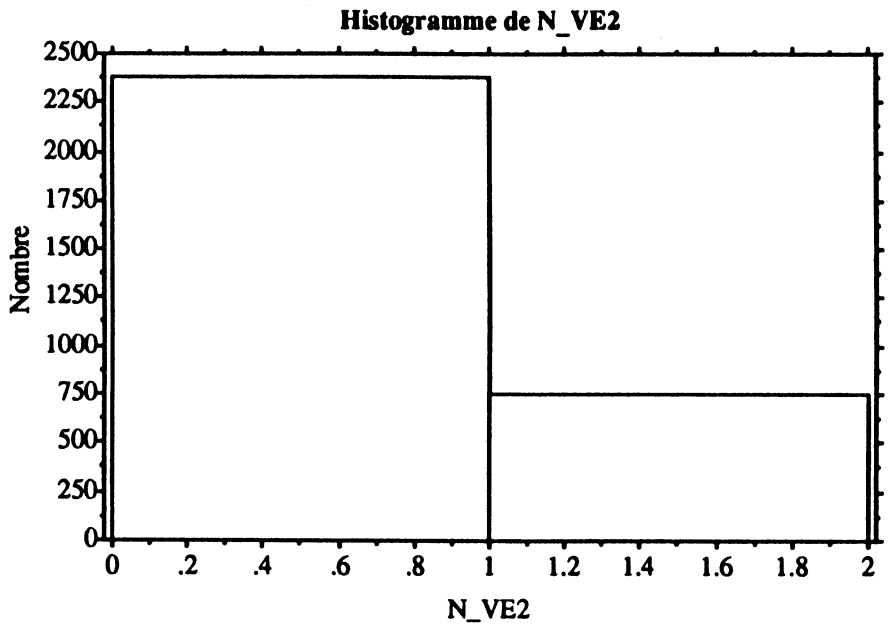


(Figure 57) Loi du nombre de versions

On peut effectivement admettre que la loi du nombre de versions successives d'une même fonction SAGA est une loi de Pascal, de paramètre estimé $\tilde{p} = 0.754$.

On peut néanmoins se poser la question de savoir s'il est judicieux de s'intéresser directement au nombre de versions successives d'une fonction, puisque les modifications apportées lors d'une nouvelle version sont susceptibles de modifier profondément la sémantique même de la fonction initiale. Nous nous sommes donc aussi intéressés aux valeurs prises par la variable qualitative indiquant simplement si la fonction à laquelle elle se rapporte sera modifiée par la suite (valeur 1) ou non (valeur 0).

Nous avons mentionné ci-dessous les valeurs des statistiques élémentaires sur cette variable, ainsi que son histogramme.

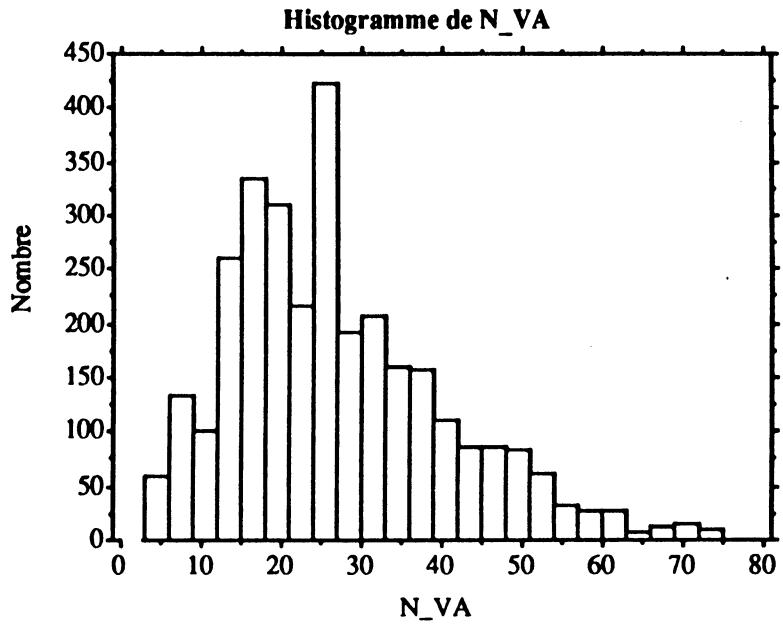


Mean:	Std. Dev.:	Std. Error:	Variance:	Coef. Var.:	Count:
.241	.428	.008	.183	177.395	3126
Minimum:	**Maximum:**	**Range:**	**Sum:**	**Sum Squared:**	**# Missing:**
0	1	1	754	754	0

(Figure 58) Etude de la modification des fonctions

C'est bien sûr l'observation d'une loi de Bernoulli. Comme observé ci-dessus, seuls 24.1 % des enregistrements collectés concernent des fonctions qui ont par la suite été modifiées. Ceci explique en partie les problèmes posés dans le paragraphe B-1-2.

9 . N_VA : Nombre total de variables accessibles dans la fonction

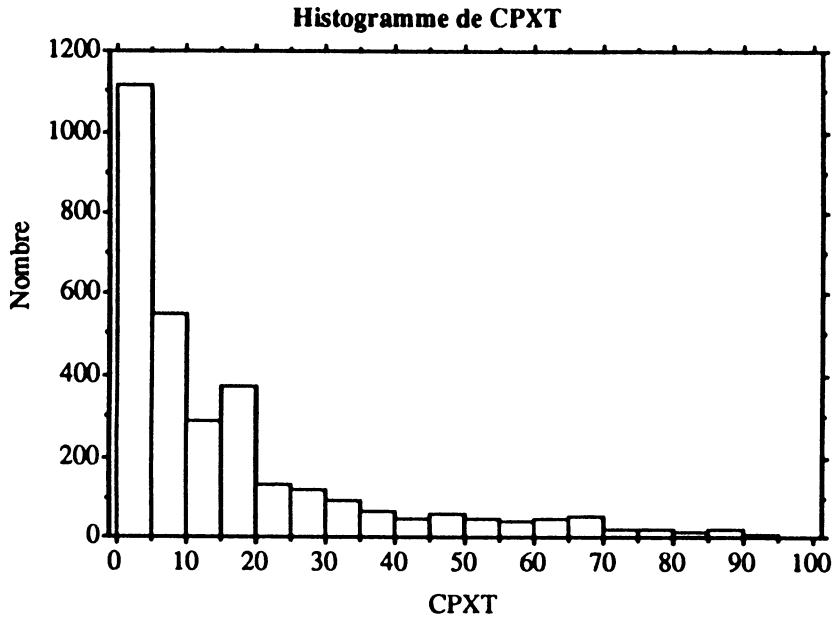


Mean:	Std. Dev.:	Std. Error:	Variance:	Coef. Var.:	Count:
26.739	14.019	.251	196.533	52.429	3126
Minimum:	**Maximum:**	**Range:**	**Sum:**	**Sum Squared:**	**# Missing:**
4	103	99	83587	2849223	0

(Figure 59) Etude du nombre de variables

Nous pouvons supposer ici que la loi suivi par cette variable aléatoire est une loi de Poisson. Mais l'adéquation est difficile.

10 . CPXT : Complexité d'assemblage de la fonction



Mean:	Std. Dev.:	Std. Error:	Variance:	Coef. Var.:	Count:
17.227	22.563	.404	509.071	130.969	3126
Minimum:	**Maximum:**	**Range:**	**Sum:**	**Sum Squared:**	**# Missing:**
0	222.026	222.026	53852.938	2518593.945	0

(Figure 60) Etude de la complexité d'assemblage

Plus d'un tiers des enregistrements collectés concernent des fonctions dont la complexité est inférieure à 5, ce qui explique en partie le peu de corrections apportées aux fonctions. Si l'on calcule la moyenne des complexités sur les fonctions soumise à future modification et sur les versions finales, on trouve :

Complexité moyenne sur les fonctions à revoir : 16.26 ;

Complexité moyenne sur les versions finales : 17.54 ,

ce qui montre que les modifications sur les fonctions ont tendance à faire croître leur complexité.

Ce résultat appelle quelques commentaires : en effet, on aurait pu s'attendre à ce que les modifications apportées à une fonction tendent à faire diminuer sa complexité. Mais il nous faut aussi tenir compte des évolutions de spécifications : c'est souvent le rajout de variables de contrôle qui provoque un accroissement significatif de la complexité d'assemblage de la fonction.

On retrouve des résultats tout-à-fait similaires à une autre échelle dans le paragraphe B-4 : la complexité d'une application croît elle aussi stochastiquement avec le temps.

B-1-2) Lien avec le nombre de versions

Nous désirons étudier ici la corrélation entre le nombre de versions d'une même fonction et les métriques associées. La matrice de corrélation des différentes observations est la suivante :

	PROF	N_VE	N_S_F	N_E	N_T	N_S	N_C	N_OP	CPXT	N_VA
PROF	1.00									
N_VE	-0.237	1.00								
N_S_F	-0.244	0.220	1.00							
N_E	-0.036	0.155	0.234	1.00						
N_T	-0.289	0.255	0.537	0.521	1.00					
N_S	-0.118	0.099	0.252	0.309	0.482	1.00				
N_C	-0.042	0.051	0.033	-0.068	0.042	0.142	1.00			
N_OP	0.077	-0.058	-0.281	-0.034	-0.175	0.076	0.510	1.00		
CPXT	0.008	-0.019	0.058	0.101	-0.105	0.210	0.483	0.824	1.00	
N_VA	-0.088	0.120	0.368	0.361	0.405	0.453	0.506	0.665	0.792	1.00
	PROF	N_VE	N_S_F	N_E	N_T	N_S	N_C	N_OP	CPXT	N_VA

(Figure 61) Matrice de corrélation sur les mesures

Le but de cette analyse était de pallier le manque de relevés d'erreurs par l'étude du nombre de versions des fonctions, en déterminant les principales caractéristiques tendant à provoquer la reprise de ces fonctions.

Au vu de cette matrice construite à partir de l'ensemble des enregistrements, nous constatons que la métrique qui explique le mieux le nombre de versions est le nombre de types. Mais attention : la corrélation correspondante n'est que de 0.25 ; il n'est donc pas possible au vu de nos mesures, de prévoir quelles fonctions vont avoir tendance à être modifiées ; la régression linéaire multiple de ce nombre de versions sur l'ensemble des métriques observées donne d'ailleurs un fort piètre coefficient d'ajustement $R^2 = 0.11$.

Mais ce résultat doit être tempéré. En effet, si l'on étudie chacune des applications séparément, on obtient alors des valeurs de coefficients de corrélation s'étageant de 0.06 à 0.65. Mieux encore ; si l'on s'intéresse uniquement à la variable N_VE2 (cf. B-1-1,8) décrivant la

modification future des fonctions, on trouve alors des coefficients de corrélation par application compris entre 0.03 et 0.73.

Il est donc possible pour certaines applications d'expliquer au vu des résultats des métriques pourquoi les fonctions la composant sont susceptibles d'être modifiées ou non, ce qui permet d'agir immédiatement sur les conceptions de ces fonctions pour éviter les reprises.

B-1-3) Progression de la complexité des fonctions

Le problème posé ici est celui du suivi de la complexité des fonctions au cours de leurs différentes versions. Vu le petit nombre de ces versions pour chacune des fonctions observées (cf. B-1-1,8), il est nécessaire de bâtir un modèle robuste.

Au vu de la quarantaine d'échantillons tels que celui présenté en annexe 2, nous allons supposer que la complexité de la $i^{ème}$ version d'une fonction dépend de sa complexité à la $(i-1)^{ème}$ version, suivant un processus de type auto-régressif.

Mais on s'aperçoit aussi qu'il arrive souvent que la complexité d'une fonction reste constante entre deux versions successives de celle-ci : ceci provient du caractère mineur de la modification, ne portant pas sur l'interface de la fonction et de ses sous-fonctions.

On définit alors la variable aléatoire δ_i telle que

$\delta_i = 1$ si l'interface de la fonction est modifiée entre les versions i et i+1 ;

$\delta_i = 0$ sinon.

Les $(\delta_i)_i$ sont bien entendu des variables aléatoires de loi de Bernoulli, que nous supposerons de paramètre constant pour toutes les fonctions de toutes les applications.

Si C_i est la complexité de la fonction à la $i^{ème}$ version, on pose alors :

$$C_i = C_{i-1} + \delta_i \cdot \varepsilon_i ,$$

où ε_i est une certaine variable aléatoire appréciant l'impact de la modification sur la complexité.

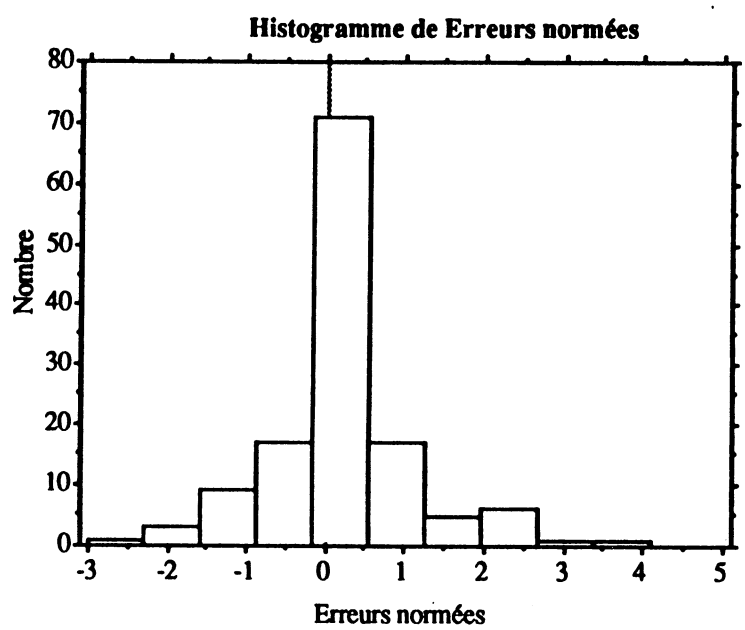
Pour pouvoir comparer les erreurs $(\varepsilon_i)_i$ entre elles, nous proposons de diviser ces erreurs par l'espérance de la complexité : si k est le nombre de versions de la fonction étudiée, alors on pose $E(C) = \frac{1}{k} \sum_{i=1}^{k} C_i$, et on introduit les nouvelles variables aléatoires ε'_i, définies par $\varepsilon'_i = \frac{\varepsilon_i}{E(C)}$.

Le paramètre de la loi des $(\delta_i)_i$ pourra aisément être calculé par l'estimateur sans biais $\frac{\text{Nombre de } \varepsilon_i \text{ différents de 0}}{\text{Nombre total de } \varepsilon_i}$: l'application numérique sur nos données nous fournit un esti-

mateur voisin de 0.188 ; c'est-à-dire que 81 % des modifications apportées sur les fonctions n'ont aucune influence sur leur complexité.

Pour étudier maintenant la loi des erreurs, il nous faut tout d'abord éliminer tous les $(\varepsilon'_i)_i$ nuls ; ce faisant, nous obtenons une nouvelle collection d'erreurs que nous noterons $(\varepsilon''_i)_i$.

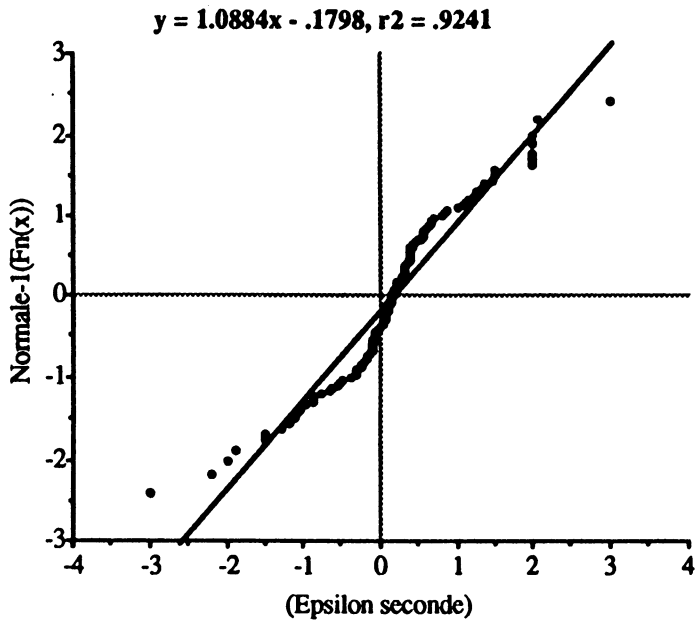
Dans la pratique, nous avons observé 131 réalisations de ε'' ; l'histogramme de ces résultats, ainsi que les statistiques usuelles, sont les suivants :



Mean:	Std. Dev.:	Std. Error:	Variance:	Coef. Var.:	Count:
.194462	.917658	.080176	.842097	471.896054	131
Minimum:	**Maximum:**	**Range:**	**Sum:**	**Sum Squared:**	**# Missing:**
-3	4	7	25.474513	114.42637	0

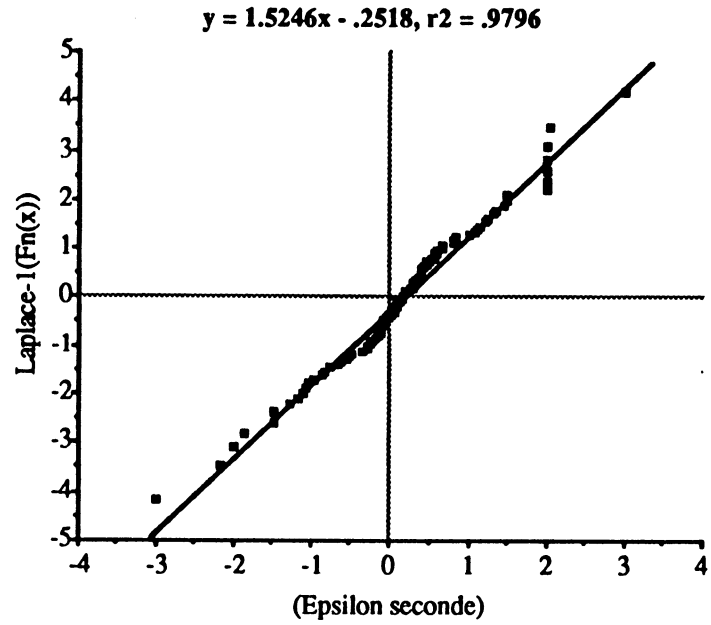
(Figure 62) Etude de la progression de la complexité d'assemblage

Nous pouvons donc supposer, au vu de l'histogramme, que la loi des ε''_i est normale. Pour vérifier cette intuition, on trace la droite de Henry associée aux observations. On obtient :



(Figure 63) Test de normalité des ε"

La régression est visiblement mauvaise. Il nous faut donc essayer une nouvelle loi de probabilité pour les ε''_i. Après avoir tenté un essai infructueux avec la loi de Cauchy, nous proposons une loi de Laplace (de densité $f(x) = \frac{1}{2\lambda} e^{-\frac{|x-\theta|}{\lambda}}$) ; on peut alors tracer l'analogue de la droite de Henry pour cette loi. On obtient le diagramme suivant :



(Figure 64) Loi des ε"

Cette fois la régression est bonne. Nous pouvons alors estimer les paramètres de la loi grâce au diagramme ; on obtient alors $\widehat{\theta} = 0.165$ et $\widehat{\lambda} = 0.656$.

Nous aurions pu également utiliser les méthodes classiques d'étude des séries chronologiques (extraction de tendance, étude de la composante saisonnière et caractérisation des résidus), mais nous avions là encore un problème concernant la taille des échantillons : aucune fonction SAGA n'a atteint plus de 6 versions. La méthode utilisée ici ne permet pas de prédire avec certitude le futur du processus, mais met à profit le grand nombre d'échantillons à notre disposition. De plus, on note que le processus des ε" est homogène pour toutes les applications, ce qui n'était pas évident au départ.

B-2) Modèle et simulation d'applications SAGA

On se propose ici de modéliser la construction de l'arbre de décomposition d'une application SAGA.

Soit Ω l'ensemble de toutes les applications SAGA possibles. On le munit d'une tribu $\mathcal{A}$ et d'une probabilité P. Il est clair que cet ensemble est infini et probablement non dénombrable, puisqu'il comprend en particulier les réponses à tous les problèmes traitables en SAGA.

Sur cet espace de départ, on définit l'application A comme suit :

A : Ω -> F

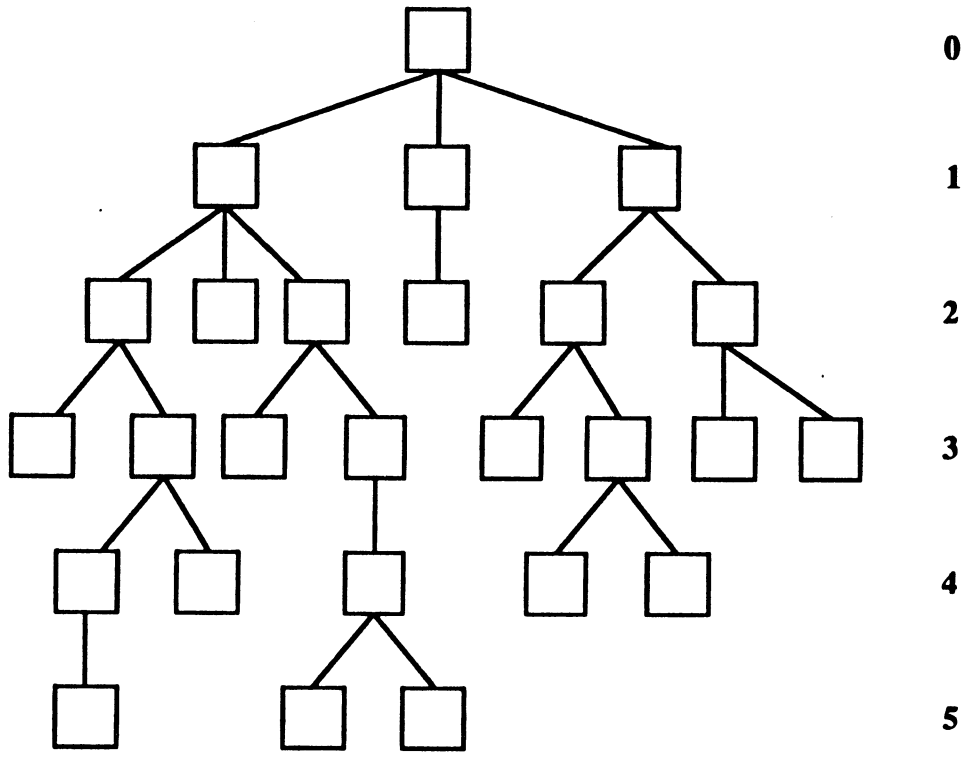
ω -> $A(\omega) = \omega'$: l'arbre de décomposition de l'application SAGA ω.

L'ensemble F apparaît comme l'ensemble de tous les arbres de décomposition possibles d'une application SAGA. Cet ensemble F est donc infini, mais dénombrable. Grâce à l'application A, on peut aussi définir la probabilité image P_A. Dans ce paragraphe, nous nous intéressons à certaines caractéristiques des arbres de décomposition d'applications SAGA.

B-2-1) Etude de l'arbre de décomposition d'une application SAGA

Nous cherchons ici à fournir un modèle probabiliste du nombre de fonctions par niveau pour l'arbre de décomposition d'une application SAGA.

Soit S_i le nombre de fonctions situées à la profondeur i dans l'arbre de décomposition d'une application SAGA quelconque.



(Figure 65) Exemple d'arbre de décomposition SAGA

Sur l'exemple ci-dessus, on a $(S_0, S_1, S_2, S_3, S_4, S_5) = (1, 3, 6, 8, 5, 3)$, et $S_i = 0$ $\forall$ i >= 6. Notre hypothèse est la suivante :

- tout se passe comme si les fonctions de profondeur i dans l'arbre pouvaient générer chacune de 0 à 6 sous-fonctions, avec une loi de probabilité binomiale $\mathfrak{B}(6, p_i)$ indépendamment les unes des autres, le paramètre de la loi n'étant lié qu'à la profondeur de la fonction.

Sous cette hypothèse, la suite des paramètres $(p_i)_i$ doit nécessairement être de limite nulle, puisqu'on ne peut avoir d'arbre de profondeur maximale infinie.

Ainsi, dans notre exemple, la fonction mère génère 3 sous-fonctions suivant une loi de probabilité $\mathfrak{B}(6, p_0)$. Chacune de ses sous-fonctions va générer indépendamment les unes des autres un certain nombre de sous-fonctions de profondeur 2 : 3 pour la première, 1 pour la seconde, et 2 pour la troisième, chacune selon une loi de probabilité $\mathfrak{B}(6, p_1)$.

Si on s'intéresse au nombre total de fonctions à la profondeur i, soit S_i, on constate que la loi de cette variable aléatoire ne dépend que du nombre de fonctions à la profondeur i-1, soit S_{i-1}. La famille des variables aléatoires $\{S_i\}_{i \in \mathbb{N}}$ est donc une chaîne de Markov. Puisque les $(p_i)_i$ sont nécessairement différents, cette chaîne de Markov n'est pas homogène.

Sous les hypothèses présentées ci-dessus, on a

$$S_i \mid S_{i-1}=s_{i-1} \text{ suit la loi } \mathfrak{B}(6\, s_{i-1}, p_{i-1}).$$

En effet, si l'on sait qu'il existe s_{i-1} fonctions au niveau i-1, et que chacune de ces fonctions génère de 0 à 6 sous-fonctions suivant une loi $\mathfrak{B}(6, p_{i-1})$, le nombre de fonctions au niveau i apparaît comme une variable aléatoire dont la loi est la somme de s_{i-1} lois binomiales $\mathfrak{B}(6, p_{i-1})$: c'est donc bien une variable aléatoire de loi $\mathfrak{B}(6\, s_{i-1}, p_{i-1})$.

On a donc :

- $S_0 = 1$;
- $S_{n+1} = \sum_{r=1}^{S_n} \zeta_{n,r}$, $\forall\, n > 0$,

où les $(\zeta_{n,r})_r$ sont une famille de variables aléatoires indépendantes dans leur ensemble, identiquement distribuées, et telles que $\zeta_{n,r}$ suit une loi $\mathfrak{B}(6,p_n)$ $\forall\, r > 0$.

La fonction génératrice de la variable $\zeta_{n,r}$ s'écrit $\varphi_n(s) = E(s^{\zeta_{n,r}}) = (p_n \cdot s + 1 - p_n)^6$. Posons $q_n = 1-p_n$. Il vient alors $\varphi_n(s) = (p_n \cdot s + q_n)^6$.

Soit ψ_n la fonction génératrice de S_n, définie par $\psi_n(s) = \sum_{k=0}^{6^n} P(S_n = k)\, s^k$. On a immédiatement $\psi_0(s) = s$, puisque $P(S_0 = 1) = 1$.

De même, on peut obtenir facilement $\psi_1(s)$; en effet :

$$\psi_1(s) = \sum_{k=0}^{6} P(S_1 = k)\, s^k = \sum_{k=0}^{6} C_6^k\, (s.p_0)^k\, q_0^{6-k} = (p_0\, s + q_0)^6 = \varphi_0(s) .$$

Le calcul du terme général est le suivant :

$$\psi_{n+1}(s) = \sum_{k=0}^{6^{n+1}} P(S_{n+1} = k)\, s^k = \sum_{k=0}^{6^{n+1}} \sum_{j=0}^{6^n} P(S_{n+1} = k \mid S_n = j) \cdot P(S_n = j)\, s^k, \text{ soit}$$

$$\psi_{n+1}(s) = \sum_{j=0}^{6^n} P(S_n = j) \sum_{k=0}^{6^{n+1}} P(\zeta_{n,1} + \zeta_{n,2} + \ldots + \zeta_{n,j} = k) \cdot s^k .$$

Or la fonction génératrice de la somme de j variables aléatoires indépendantes de même loi s'exprime comme la puissance j de la fonction génératrice de chacune de ces variables. On a donc $\sum_{k=0}^{6^{n+1}} P(\zeta_{n,1} + \zeta_{n,2} + \ldots + \zeta_{n,j} = k) \cdot s^k = (p_n \cdot s + q_n)^{6j}$, et par suite

$$\psi_{n+1}(s) = \sum_{j=0}^{6^n} P(S_n = j)\, (p_n \cdot s + q_n)^{6j} = \psi_n(\varphi_n(s)) .$$

En réitérant l'opération, on obtient

$$\psi_{n+1}(s) = \psi_{n-1}(\varphi_{n-1}(\varphi_n(s))) = \psi_{n-k}(\varphi_{n-k}(\varphi_{n-k+1}(\ldots(\varphi_n(s))\ldots)) \ \forall\ k = \{1,2,\ldots,n-1\}.$$

Pour $k = n-1$, on obtient $\psi_{n+1}(s) = \psi_1(\varphi_1(\varphi_2(\ldots(\varphi_n(s))\ldots)) = \varphi_0(\varphi_1(\varphi_2(\ldots(\varphi_n(s))\ldots))$. On a en résumé :

$$\psi_0(s) = s.$$
$$\psi_n(s) = \varphi_0(\varphi_1(\varphi_2(\ldots(\varphi_{n-1}(s))\ldots)) \ \forall\ n > 0$$

On peut désormais calculer l'espérance mathématique de S_n. En effet, on a $E(S_n) = \psi'_n(1)$. Le calcul nous donne :

$$\psi'_n(s) = \varphi'_0(\varphi_1(\varphi_2(\ldots(\varphi_{n-1}(s))\ldots)) \cdot \varphi'_1(\varphi_2(\ldots(\varphi_{n-1}(s))\ldots) \ldots \varphi'_{n-2}(\varphi_{n-1}(s)) \cdot \varphi'_{n-1}(s) .$$

Or $\varphi'_k(s) = 6\,(\,p_k \cdot s + q_k\,)^5 \cdot p_k$, donc $\varphi'_k(1) = 6\,p_k$,
et $\varphi_k(1) = 1$.

Donc $\psi'_n(1) = 6\,p_0 \cdot 6\,p_1 \ldots 6\,p_{n-1}$, et

$$E(S_n) = 6^n \prod_{k=0}^{n-1} p_k .$$

La moyenne de S_n tend donc rapidement vers 0, puisque la suite $(p_k)_k$ tend vers 0.

B-2-2) <u>Etude de l'extinction de l'arbre de décomposition</u>

Nous nous proposons maintenant d'étudier la variable aléatoire "Profondeur maximale" d'une application SAGA. On définit donc une nouvelle application N par :
N : F -> $\mathbb{N}$
ω' -> N(ω') : profondeur de l'arbre de décomposition ω'.

Pour ce faire, on pose en préliminaire

$$\alpha_n = P(\,S_n = 0\,) = \psi_n(0) = \varphi_0(\varphi_1(\varphi_2(\ldots(\varphi_{n-1}(0))\ldots)).$$

On a :

* $\varphi_{n-1}(0) = q_{n-1}{}^6$;
* $\varphi_{n-2}(\varphi_{n-1}(0)) = (p_{n-2} \cdot q_{n-1}{}^6 + q_{n-2})^6$;
* $\varphi_{n-3}(\varphi_{n-2}(\varphi_{n-1}(0))) = (p_{n-3} \cdot (p_{n-2} \cdot q_{n-1}{}^6 + q_{n-2})^6 + q_{n-3})^6$;

Par récurrence, on a $\alpha_n = (p_0 (p_1 (\ldots (p_{n-3} \cdot (p_{n-2} \cdot q_{n-1}{}^6 + q_{n-2})^6 + q_{n-3})^6 + \ldots + q_0)^6$.

Pour calculer la loi de N, on peut utiliser le fait que $P(S_{n+1} = 0) = P(N \leq n)$. On a alors :

$$P(N = n) = P(N \leq n) - P(N \leq n-1) = \alpha_{n+1} - \alpha_n .$$

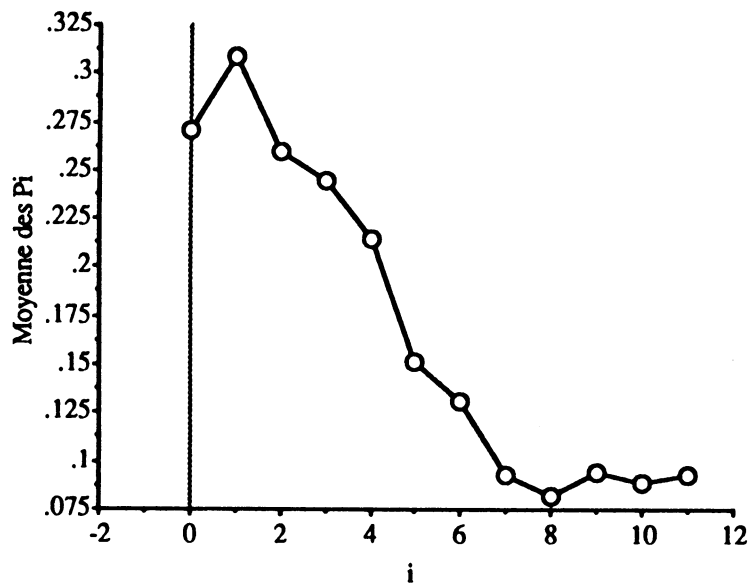
On constate ici que grâce à ces calculs et aux résultats fournis dans le paragraphe (B), il est possible de simuler le développement des applications SAGA.

B-3) Etude de la morphologie des logiciels produits

Nous cherchons ici à vérifier la justesse du modèle présenté ci-dessus. Pour ce faire, nous allons calculer des statistiques sur les applications à notre disposition.

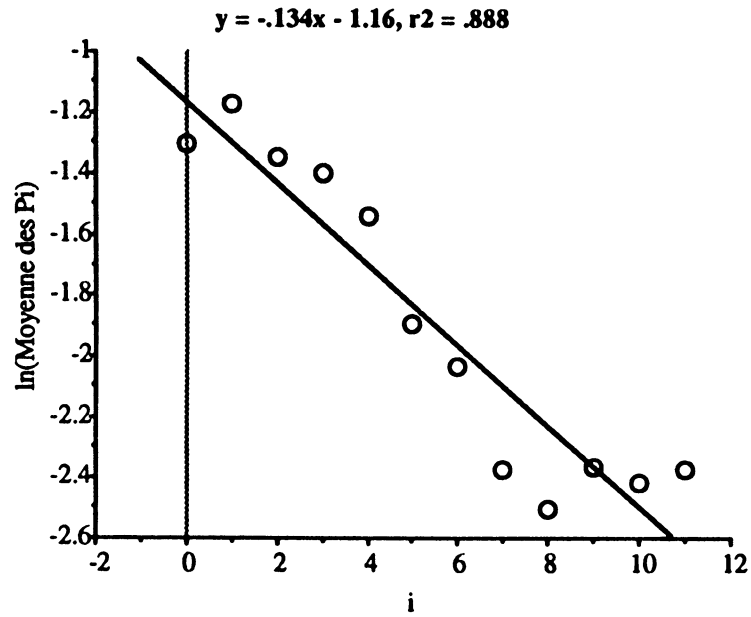
Pour chaque application, nous allons estimer p_i par $\tilde{p}_i = \frac{\tilde{S}_{i+1}}{\tilde{S}_i}$, où $\tilde{S}_i$ désigne le nombre de fonctions observées pour l'application au niveau i. Bien entendu, si la profondeur de l'application est $\tilde{N}$, on aura $\tilde{p}_{\tilde{N}} = 0$, et tous les $\tilde{p}_j$ seront inconnus pour $j > \tilde{N}$.

Puis on estime la valeur finale de p_i par la moyenne des $\tilde{p}_i$ trouvés. On peut alors tracer la courbe des valeurs de p_i en fonction de i ; on obtient :



(Figure 66) Estimation des $(p_i)_i$

Au vu de cette courbe, nous allons tester l'hypothèse suivant laquelle les $(p_i)_i$ décroissent géométriquement, c'est-à-dire $p_{i+1} = c \cdot p_i$, soit $p_i = p_0 \cdot c^i$. Un moyen consiste à calculer la régression linéaire du logarithme des p_i sur i. La courbe obtenue est alors la suivante :



(Figure 67) Loi des $(p_i)_i$

L'adéquation est bonne pour les valeurs comprises entre 1 et 6 ; on peut alors admettre que le nombre de sous-fonctions par fonction suit bien une loi binomiale $\mathfrak{B}(6, p_i)$, dont le coefficient p_i ne dépend que du niveau de la fonction dans l'arbre de décomposition de l'application. En effet, tous les points figurant sur le diagramme ci-dessus n'ont pas été estimés avec la même précision : si 90% des applications observées ont une profondeur maximale supérieure ou égale à 2, ce qui permet d'estimer p_2 à l'aide de 30 observations, seulement 10% d'entre elles sont de profondeur maximale supérieure ou égale à 9, ce qui nous laisse simplement 6 observations pour estimer p_9.

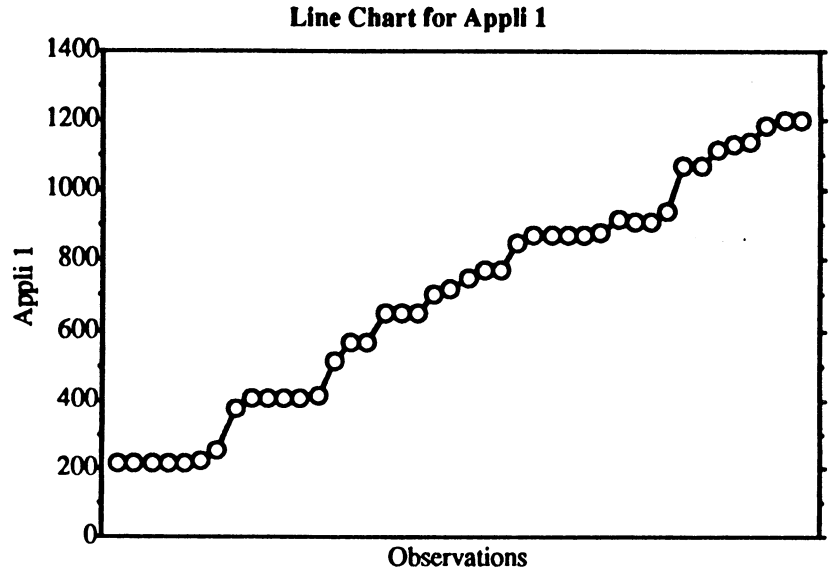
B-4) Etude de l'évolution de la complexité d'une application SAGA

Parallèlement aux mesures sur les fonctions modifiées en cours de journée, nous avons mesuré chaque jour l'évolution de la complexité globale des applications, comme définie dans le chapitre 3, paragraphe D. Cette collecte a porté sur les 38 applications qui s'y prêtaient le mieux. Les résultats que nous avons obtenus se présentent donc sous la forme d'une liste par application comportant la suite des valeurs prises par la complexité globale de l'application au cours du temps.

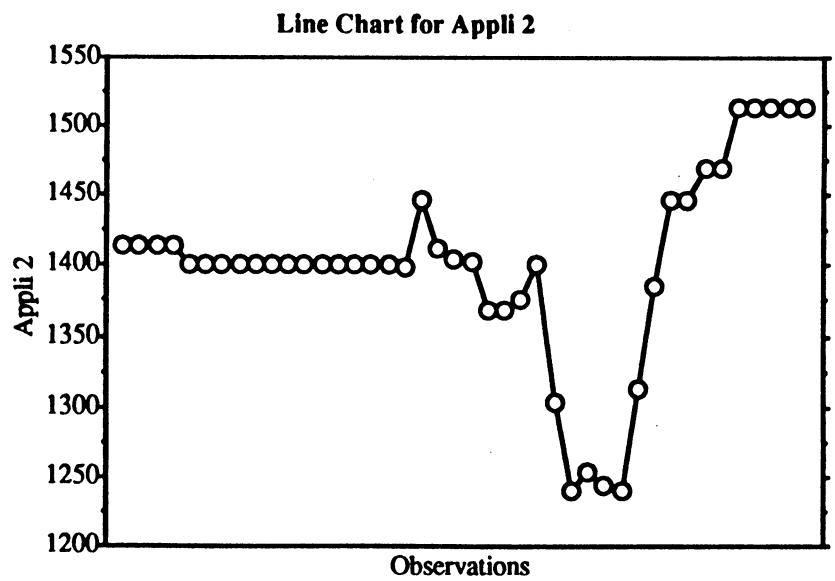
Malheureusement, il se trouve que très peu d'applications ont été suffisamment modifiées pendant ces mesures pour que le résultat soit significatif.

Néanmoins, nous avons sélectionné les graphiques correspondants à 9 applications intéressantes de ce point de vue. Nous présentons ici ces diagrammes d'évolution de la

complexité globale de ces applications du 10 mai au 10 juillet, soit 42 mesures pour chaque application.

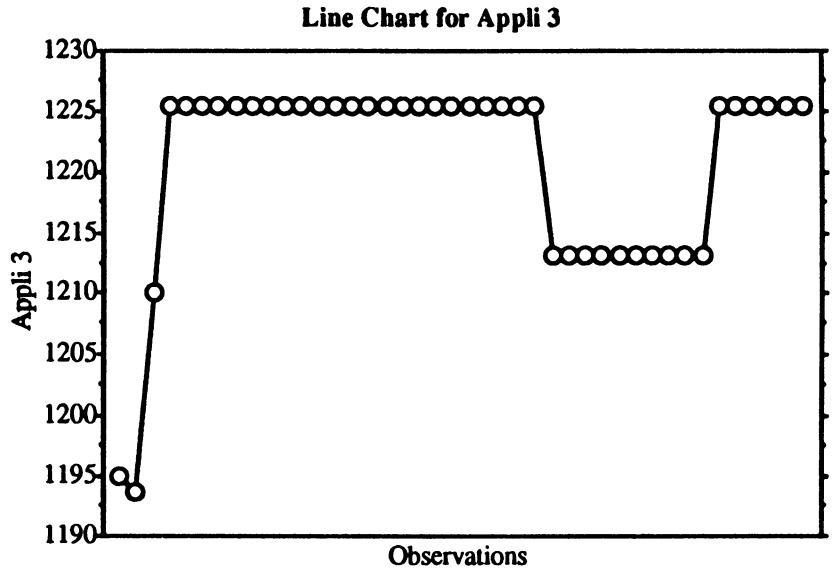


Nous avons ici affaire à une application dont les modifications ont entraîné une progression régulière de la complexité. C'est un cas typique de rajout de variables de contrôle à l'intérieur des fonctions composant l'application.

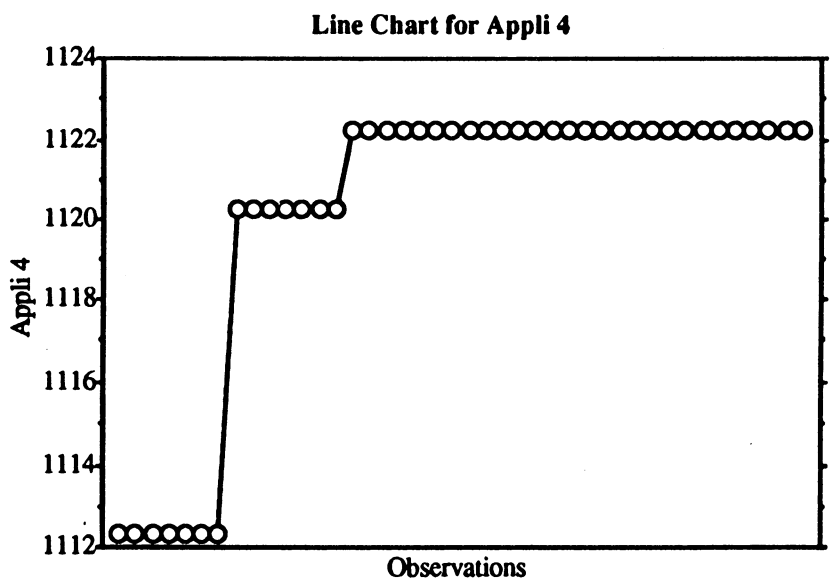


Cette deuxième application nous montre un autre aspect du développement d'une application SAGA : dans un premier temps, les modifications apportées sont minimes, puis on supprime une des fonctions de l'application, entraînant une diminution notable de la

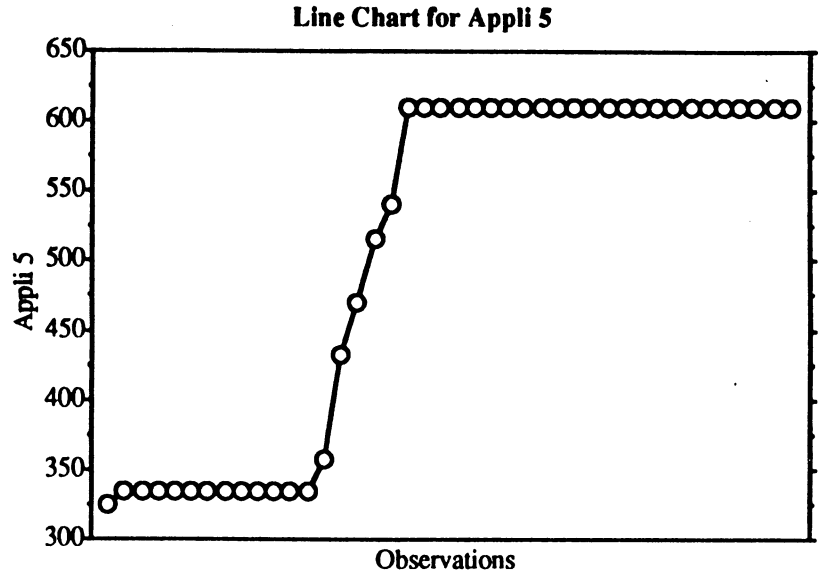
complexité ; enfin, on rajoute une nouvelle fonction de taille légèrement supérieure à la fonction supprimée, de sorte que la complexité totale de l'application a globalement augmenté.



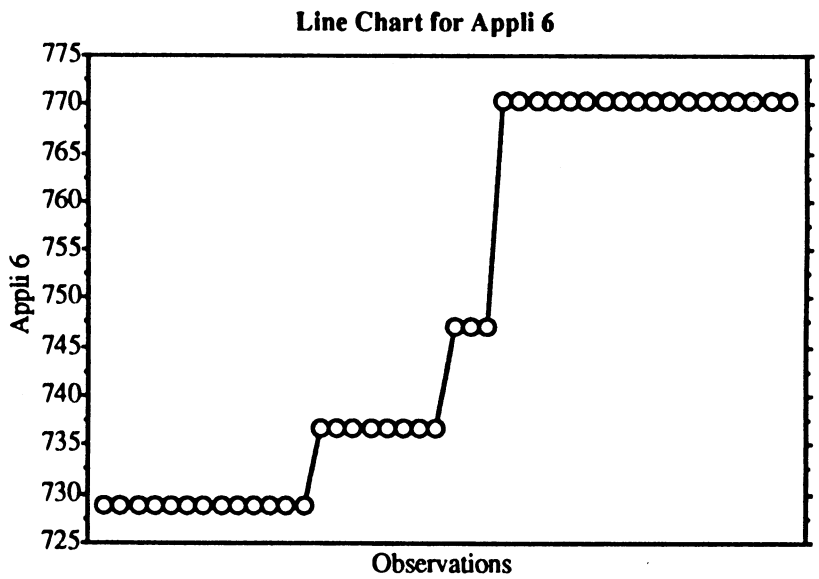
Ici, on observe une application dont toutes les modifications ont très peu d'effets sur la complexité globale : celle-ci reste toujours comprise entre 1190 et 1225.



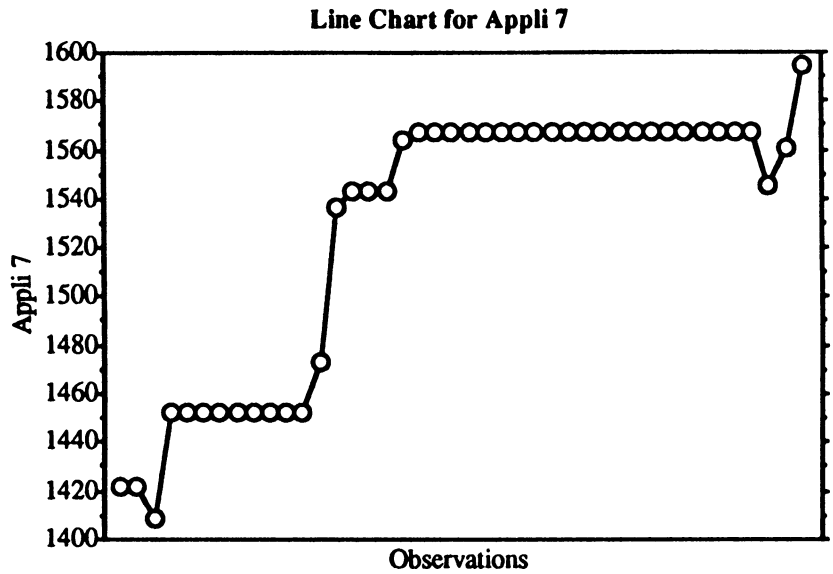
Là encore, très peu de modifications significatives de la complexité de l'application.



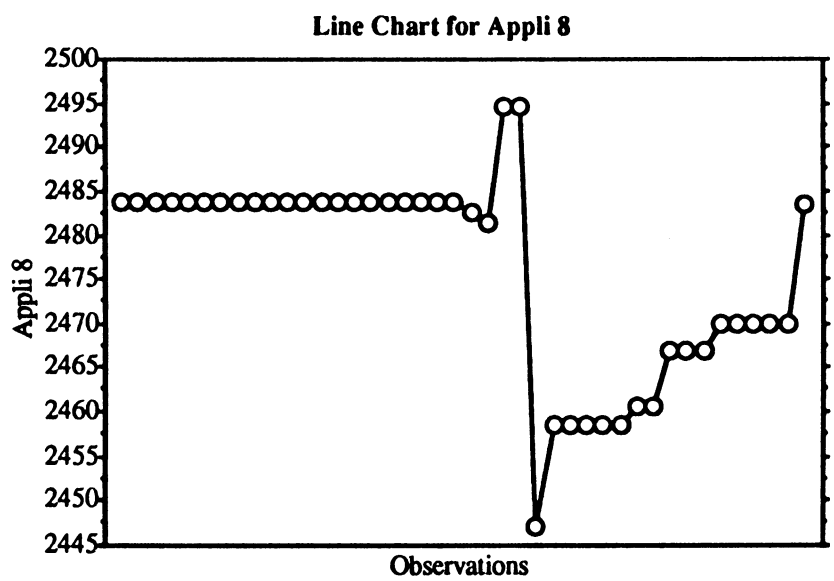
Sur ce schéma, on peut observer une application à laquelle on a rajouté en cours de développement un nombre de fonctions équivalent à celui qu'elle avait au départ ; la complexité globale s'en trouve pratiquement doublée.



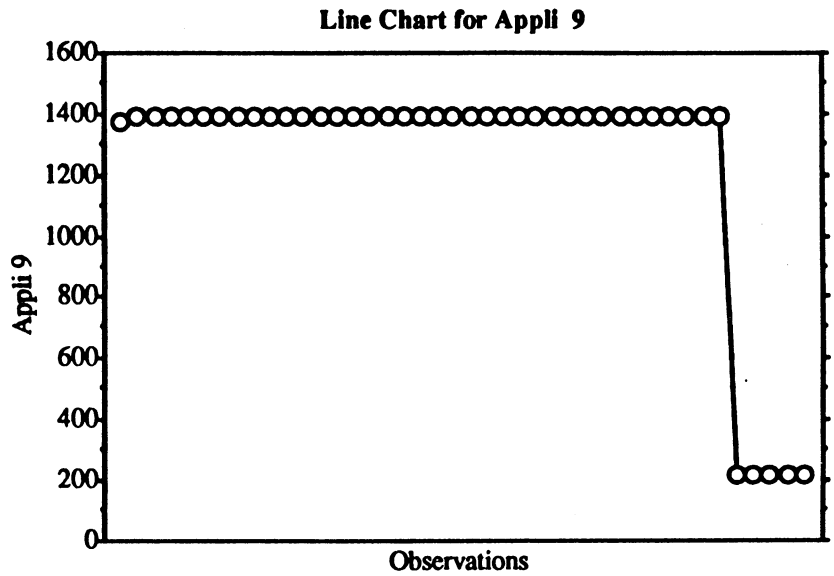
La progression de la complexité globale de la sixième application, quoique positive, reste très faible : on passe ainsi de 730 à 770.



On voit là encore l'effet du rajout de fonctions sur la complexité globale de l'application.



Ici, comme pour la deuxième application étudiée, on constate une suppression de fonction, suivie du rajout d'une fonction de taille comparable. Le résultat est équivalent à celui obtenu pour l'application numéro 2.



Sur ce neuvième et dernier schéma, on peut constater l'effet de la suppression de tout un pan de l'application sur la complexité globale de l'application : il s'agissait ici d'une série de contrôles nécessaires pour vérifier la cohérence de plusieurs applications entre elles, mais qui n'avaient plus de raison d'être une fois ces vérifications terminées.

En résumé, on retrouve ici des résultats similaires à ceux obtenus dans le paragraphe B-1-1 : les modifications apportées aux diverses fonctions composant l'application sont le plus souvent des modifications de spécifications (rajout de variables dans les fonctions), ce qui tend à faire augmenter la complexité globale de l'application.

Etant donné la diversité des résultats obtenus ici, il ne nous a pas semblé nécessaire de mener une étude statistique approfondie sur ces courbes. Les diagrammes présentés ci-dessus sont rapportés à titre indicatif.

CONCLUSION

Cette étude présente une quantification possible de la complexité du logiciel dans le cas d'un langage de type "flots de données". La mesure de complexité proposée est compatible avec l'axiome de la systémique suivant lequel "la complexité d'un système est supérieure à la somme des complexités des sous-systèmes le composant". Nous avons utilisé l'aspect hiérarchique des applications étudiées, ainsi que les propriétés d'assemblage des fonctions composant ces applications : c'est en effet sur l'échange de données entre ces fonctions que notre métrique est basée. Cette proposition constitue un apport original concernant la complexité des logiciels, puisqu'à notre connaissance aucune étude de ce type n'avait encore été menée sur des langages de type "flots de données". Une campagne de mesures nous a fourni la plage de variation de la métrique proposée sur des applications réelles, et nous avons exposé les liens qu'elle présente avec d'autres mesures de complexité applicables à cette catégorie de logiciels.

Le travail présenté ici n'est certes pas exhaustif du point de vue de la fiabilité. En particulier, on pourrait s'intéresser à la modélisation mathématique de la relation qui existe sans doute entre la complexité des applications et leur fiabilité. Des études empiriques ont d'ores et déjà été menées sur des langages usuels (voir [ALB-82] par exemple), mais jamais dans un contexte "flots de données". Mais les logiciels qui ont fait l'objet de mesures pour ce travail sont conçus pour rester opérationnels pendant 25 ans ; il nous aurait été difficile d'attendre des résultats de défaillance sur leur vie opérationnelle, d'autant plus que ces défaillances sont nécessairement très peu nombreuses : sur l'ensemble des 10 versions précédentes du même projet, on avait détecté 6 erreurs (minimes d'ailleurs) pour 50000 lignes de code en 5 ans d'exploitation !

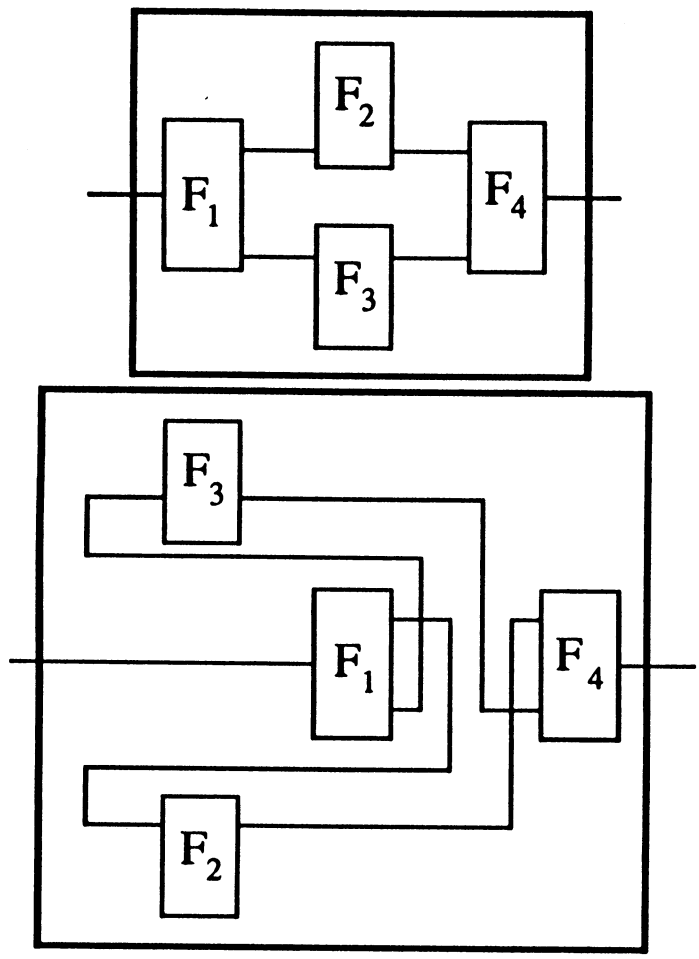
Toutefois, la mesure de complexité proposée ici est intimement liée à la fiabilité, puisqu'elle est basée sur le nombre de relations possibles entre les fonctions hors typage des variables, et donc au nombre de façons possibles de se tromper dans les connexions. D'autre part, nous avons procédé à une campagne de mesures dont l'étude des résultats fournit des enseignements utiles quant à la morphologie des applications SAGA, ainsi qu'à leur comportement vis-à-vis de la complexité, et donc de la fiabilité.

Pour en terminer avec les remarques sur la fiabilité, il nous faut conclure qu'au vu de notre échantillon il n'existe pas de relation statistique entre le nombre de versions successives d'une fonction et la suite des valeurs prises par sa complexité. Pour compléter ce travail, il faudrait étudier soigneusement la relation entre le nombre de fautes trouvées durant les phases de test et d'exploitation, et la complexité des fonctions. Mais d'ores et déjà, on peut supposer que ces relations seront difficiles à établir.

Il nous faut souligner que l'étude locale de la complexité sur chacune des fonctions n'a été possible que parce que le langage SAGA ne permet pas d'effets de bord : en effet, SAGA n'admet pas la définition de variables globales. Nous pensons que, moyennant certains aménagements, cette approche pourrait être appliquée à d'autres systèmes possédant cette particularité, puisque la définition de la complexité donnée dans le troisième chapitre est valable pour tout système hiérarchique comportant ou non une notion de type : en fait, il nous semble possible d'appliquer la même démarche à tout système hiérarchique fonctionnel tel que par exemple les langages fonctionnels. Nous espérons en particulier que la métrique présentée ici pourra être testée effectivement sur d'autres langages de type "flots de données" (comme LUSTRE ou SIGNAL), et les résultats confrontés à ceux rapportés dans ce document.

Il serait aussi intéressant d'essayer d'adapter la métrique de Halstead aux logiciels SAGA, par exemple en utilisant cette métrique sur la traduction correspondante du programme en LUSTRE, (LUSTRE est un langage de type "flots de données" textuel englobant SAGA).

Enfin, nous terminons par la remarque suivante : l'aspect graphique du langage SAGA induit une complexité supplémentaire dans les vues, complexité que l'on peut qualifier de subjective. Dans les langages de type "flots de données" textuels, l'ordre des équations n'a pas grande importance ; mais dans un langage graphique tel que SAGA, l'agencement des sous-fonctions dans la représentation peut influer sur la compréhension de la fonction. Considérons en effet l'exemple de la figure 68.



(Figure 68) Deux représentations d'une même fonction

Les deux configurations présentées sont strictement identiques du point de vue de leur sémantique : elles représentent la même fonction. Mais on constate que celle du bas est sensiblement plus difficile à comprendre que celle du haut. A la fin du paragraphe A du troisième chapitre, nous avions soulevé la remarque suivant laquelle la complexité d'une fonction SAGA dépendait vraisemblablement de nombreux facteurs ; il nous semble plus précisément ici que la complexité d'une fonction SAGA dépend en grande partie de deux termes : l'un lié au graphe de connexions comme présenté dans ce document, et l'autre dépendant de facteurs plus **subjectifs** comme le placement des sous-fonctions sur la représentation graphique, le routage des flots de données dans la vue, ou encore les croisements d'arcs du flot de données à l'intérieur de la fonction.

Il est toutefois clair que toute mesure de complexité ne peut et ne doit être qu'une aide à la conception des programmes, à leur vérification et à leur test, et non un outil de censure. Les mesures de complexité doivent aider les programmeurs et les vérificateurs à focaliser leur

attention sur certaines fonctions plus que sur d'autres, mais pas à remplacer ni ces personnes, ni leurs tests.

ANNEXE 1 - PROGRAMMES

Nous nous proposons ici de présenter et de commenter les structures de données et les algorithmes choisis pour mettre en œuvre le calcul de la complexité d'assemblage. Cette annexe sera divisée en 3 parties traitant respectivement des structures de données, des calculs de base nécessaires, et enfin du calcul à proprement parler. Les algorithmes seront présentés en C.

STRUCTURES DE DONNEES

Pour la lecture de la matrice de connexions, nous avons bien entendu choisi dans un premier temps un tableau à double entrée, que nous nommons MATRICEVUE ; immédiatement, on réunit les variables de même type de cette matrice, et on forme la matrice équivalente triée MATRICE_EQ. Puis on choisit un mode de représentation par tableau de blocs : chaque bloc (de type "bloc") est représenté dans ce tableau par son nombre de producteurs P, son nombre de consommateurs C, son nombre de familles n, et l'ensemble de ses sous-blocs nuls, chacun de ceux-ci étant repéré par deux entiers "c" et "p" correspondant au couple (c_i, p_i). C'est le type "tableaudeblocs".

Il nous faut à présent modéliser le vecteur de connexions $\gamma = (\gamma_0, \gamma_1, \ldots, \gamma_{n-1})$ (cf. § C-3-2). Pour ce faire, nous avons choisi une structure "etat" composée d'un entier "n" et d'un tableau "T" des différents nombres de consommateurs affectés par famille.

Enfin, puisque la formule donnée dans le paragraphe C-3-3 est récursive, il nous a fallu stocker l'ensemble des résultats intermédiaires pour éviter de répéter des calculs : cette structure, STOCK_BLOCKS, se présente comme un tableau à 4 dimensions, dans lesquelles nous allons stocker P, C, nsb, et l'indice repérant la description du premier sous-bloc dans le tableau des sous-blocs "pc" (indirection). Les valeurs de ce tableau STOCK_BLOCKS sont rangées suivant un ordre croissant défini sur les blocs, par l'intermédiaire de leurs indices : ceux-ci sont en effet classés dans un arbre binaire dont la gestion est équilibrée [WIR-86, p. 218].

L'ensemble de ces structures de données a été choisi essentiellement pour des raisons de rapidité d'exécution des algorithmes et de place mémoire. En langage C, nous obtenons :

```
/*---------------------------- DECLARATIONS ----------------------------*/

#define NbvarmaX 100            /* Nombre maximum de variables possibles par vue */
#define MemoirE 80000           /* Taille des tableaux de stockage des résultats */
#define MemoirE25 600000        /* Taille du tableau des sous-blocs associés */
#define nil 0

/*=============================================*/
/*==========          TYPES          ==========*/
/*=============================================*/

typedef int matrice01[NbvarmaX+1][NbvarmaX+1] ;  /* Lecture de la matrice initiale */

typedef struct {    /* Modélisation d'une famille de variables */
      int p, c ;  /* p : hauteur (producteurs) ; c : largeur (consommateurs) */
} couple ;

typedef struct {    /* Modélisation d'un bloc */
      int P, C, nsb ;     /* Modélisation globale du bloc :
                                 P   : nombre de producteurs du bloc ;
                                 C   : nombre de consommateurs du bloc ;
                                 nsb : nombre de familles représentées */
      couple SB[NbvarmaX+1] ;  /* Liste des familles, de 1 a nsb */
} bloc ;

typedef bloc tableaudeblocs[NbvarmaX+1] ;
      /* Modélisation d'une vue SAGA par liste des blocs */

typedef struct {          /* Mémorisation du vecteur de connexions */
      int n ;         /* Nombre total de familles du bloc traité */
      int T[NbvarmaX+1] ;  /* Nombre de consommateurs affectés dans chacune des
                                 familles du bloc courant. Ex : si T = (1,2,1,3),
                                 avec etat.n = 3, alors on a affecté :
                           - T[0] = 1 consommateur parmi les consommateurs libres,
                           - T[1] = 2 consommateurs de la première famille,
                           - T[2] = 1 consommateur de la deuxième famille,
                           - T[3] = 3 consommateurs de la troisième famille*/
} etat ;
```

```
typedef struct s_indice {       /* Classement des indices des blocs déjà calculés */
        int i ;
        int equilibre ;
        struct s_indice *droite, *gauche ;
} indice ;
typedef indice *ptindice ;

/*================================================*/
/*==========          VARIABLES          ==========*/
/*================================================*/

tableaudeblocs BLOCSVUE ;       /* Représentation de la matrice par blocs */
matrice01 MATRICEVUE ;          /* Matrice des connexions possibles dans la vue courante */
matrice01 MATRICE_EQ ;          /* Matrice en 0-1 équivalente, triée */
int NCONS ;                     /* Nombre de consommateurs de la vue */
int NPROD ;                     /* Nombre de producteurs de la vue */
int NOBLOCS ;                   /* Nombre de blocs de la matrice */

/*------------------------------ FIN DES DECLARATIONS ------------------------------*/
```

Nous supposons maintenant que la matrice à traiter se trouve dans la variable MATRICE_EQ définie précédemment. Nous définissons alors la variable BLOCSVUE contenant la matrice de connexions sous la forme de blocs.

CALCULS DE BASE

Nous définissons dans cette partie l'ensemble des fonctions de calcul élémentaires pour notre problème. Ainsi, nous supposons pouvoir disposer des fonctions suivantes :

- void FACTORIELLE(n, pres), qui renvoie sous le pointeur "pres" le résultat de n! ;
- void Cnp(n, p, pres), qui renvoie sous le pointeur "pres" le résultat de C_n^p ;
- int EXIST_BLOC(pB, presultat), qui renvoie sous le pointeur "presultat" la valeur du nombre de graphes associés au bloc repéré par son pointeur "pB" si celle-ci a déjà été calculée, ou qui renvoie 0 sinon ;
- void RANGER_BLOC(pB, val), qui range le bloc calculé repéré par son pointeur "pB" dans la structure de stockage des blocs avec la valeur du nombre de graphes associé "val" ;
- void F00(C, P, pres), qui renvoie sous le pointeur "pres" la valeur du nombre de graphes associé à un bloc simple.

CALCUL DE LA COMPLEXITE D'ASSEMBLAGE

C'est le but de la fonction NOMBREDEGRAPHES(pB, pres), où "pB" est la pointeur repérant le bloc à traiter et "pres" celui du résultat du calcul.

Mais avant d'aborder la fonction NOMBREDEGRAPHES, il est nécessaire de s'attarder sur les différentes autres fonctions dont nous avons besoin ici.

- void MAXSTATES(pMXST, pB) : "pMXST" est un pointeur sur une structure de type "etat", et "pB" est un pointeur sur le bloc à traiter ; cette fonction renvoie sous le pointeur "pMXST" les valeurs maximales des composantes du vecteur de connexions correspondant au bloc repéré par "pB".
- void INITSTATE(pCS, pMX, n) : "pCS" et "pMX" sont des pointeurs sur une structure de type "etat", et "n" est un entier ; cette fonction renvoie sous le pointeur "pCS" la première valeur du vecteur de connexions γ défini grâce au pointeur "pMX" initialisé dans la fonction précédente, et à l'entier "n" repérant le nombre total de consommateurs à connecter.
- void SUCCSTATE(pCURRSTATE, pMXST, pSTOP) ; "pCURRSTATE" et "pMXST" sont des pointeurs sur une structure de type "etat", et "pSTOP" est un pointeur sur un entier ; cette fonction renvoie dans "pCURRSTATE" la valeur suivante du vecteur de connexions γ. "pSTOP" est basculé à 1 si tous les vecteurs de connexions possibles ont été générés.

- void NEWBLOCK(pCS, pB, pNWB, n) : "pCS" est un pointeur sur une structure de type "etat", "pB" et "pNWB" sont des pointeurs sur une structure de type "bloc", et "n" est un entier ; cette fonction renvoie sous le pointeur "pNWB" la structure du bloc équivalent restant après connexion des "n" consommateurs choisis.

- int CORRECT(pB) ; cette fonction renvoie 1 ssi le bloc repéré par le pointeur "pB" est réalisable.
- void ORDONNER(pB) ; cette fonction trie les sous-blocs nuls du bloc repéré par le pointeur "pB" dans l'ordre décroissant.
- void NOMBREDECOMBINAISONS(pCS, pMX, pres) ; "pCS" et "pMX" sont des pointeurs sur une structure de type "etat", et "pres" est un pointeur sur un réel double ; cette fonction renvoie sous "pres" le nombre de combinaisons possibles pour choisir les consommateurs ayant même vecteur de connexions.

L'écriture de la fonction NOMBREDEGRAPHES en C donne le résultat suivant :

```
void NOMBREDEGRAPHES( pB, pres )
    bloc *pB ;
    double *pres ;
/* Calcul du nombre de graphes réalisables sur le bloc B, supposé vérifié correct et ordonné */
{
    double NGcour, NComb, inter ;
            /*  NGcour : valeur courante de NOMBREDEGRAPHES ;
                NComb  : nombre de graphes possibles pour la configuration
                         courante du bloc équivalent restant ;
                inter  : valeur de NOMBREDEGRAPHES pour un bloc équivalent
                         restant (appel récursif) */
    int HDB, LDB, nbiter, n, i ; /*  HDB   : Hauteur du Dernier sous-Bloc ;
                                     LDB   : Largeur du Dernier sous-Bloc ;
                                     nbiter : nombre maximum de consommateurs que
                                         l'on peut connecter */
    bloc NWB ;      /* Bloc partiel courant de l'itération principale */
    etat MXST, CURRSTATE ; /* MXST     : nombre maximum de consommateurs que
                              l'on peut affecter dans chacun des sous-blocs, y
                              compris dans les consommateurs libres ;
                          CURRSTATE : nombre de consommateurs affectés dans
                              chacun des sous-blocs de B pour l'état courant */
    int arret ;  /* Basculé a "1" après étude du dernier bloc partiel construit à partir de B */

    if (pB->nsb==0)        /* Il n'y a qu'une famille : on appelle F00 */
        F00(pB->C,pB->P,&NGcour) ;
    else  if (EXIST_BLOC(pB,&NGcour)==0)   /* Il nous faut alors le calculer */
        {     /* Ici, on a au moins une famille : on raisonne par rapport au nombre de
              consommateurs connectes au dernier producteur de la dernière famille */
              HDB = pB->SB[pB->nsb].p ;       /* Hauteur du Dernier sous-Bloc */
              LDB = pB->SB[pB->nsb].c ;       /* Largeur du Dernier sous-Bloc */
                  /* Calcul du nombre maximum de consommateurs affectables au
                       dernier producteur */
              if (HDB+LDB<pB->P)    nbiter = (pB->C)-(pB->P)+1 ;
                                 else nbiter = (pB->C)-HDB-LDB+1 ;
                  /* Détermination du nombre de consommateurs de chaque famille
                  du bloc B, et du nombre de consommateurs libres */
              MAXSTATES(&MXST,pB) ;
                  /* Initialisation de la valeur courante de NGcour */
```

```
        NGcour = 0 ;
        for ( n=1 ; n<=nbiter ; n++ ) /* On sélectionne "n" consommateurs pour
                                         le dernier producteur */
        {
                /* Initialisation de l'état courant */
            INITSTATE(&CURRSTATE,&MXST,n) ;
            arret = 0 ;
            while (arret==0)
            {
                /* Itération sur toutes les affectations de "n" consommateurs possibles,
                   représentées par CURRSTATE ; création du bloc équivalent
                   correspondant à l'état CURRSTATE à partir du bloc B */
                NEWBLOCK(&CURRSTATE,pB,&NWB,n) ;
                if (CORRECT(&NWB)==1)
                {
                   /* Si le nouveau bloc a déjà été traité, on se contente d'aller cher-
                      cher le résultat (renvoyé par la fonction EXIST_BLOC) ; sinon,
                      on appelle récursivement la procédure NOMBREDEGRAPHES */
                      ORDONNER(&NWB) ;
                      if (EXIST_BLOC(&NWB,&inter)==0)
                              /* Le bloc n'a pas déjà été calculé */
                          NOMBREDEGRAPHES(&NWB,&inter) ;
                      NOMBREDECOMBINAISONS(&CURRSTATE,
                                           &MXST,&NComb) ;
                          /* Mise à jour de NGcour */
                      NGcour = NGcour + NComb * inter ;
                }
                    /* Détermination du nouvel état à étudier : s'il n'y a
                    plus de nouvel état possible (cas où l'on a étudié
                    tous les états), "arret" est basculé à  "1" */
                SUCCSTATE(&CURRSTATE,&MXST,&arret) ;
            }
        }
        RANGER_BLOC(pB,NGcour) ;  /* Il faut alors ranger le résultat du calcul */
    }
    *pres = NGcour ;  /* Il ne reste plus qu'à affecter à la fonction sa valeur */
}
```

ANNEXE 2 - EXEMPLE DE FICHIER

Nous présentons ici un exemple de fichier de résultat de la campagne de mesures. Dans l'ordre, on peut lire pour chaque enregistrement le nom de la fonction concernée, le nombre de sous-fonctions de cette fonction, son nombre d'opérateurs, d'entrées, de sorties, de constantes utilisées, de variables, de types de variables, sa complexité d'assemblage, la date de l'enregistrement (en quantième), sa profondeur dans l'arbre de décomposition de l'application, le nombre de versions futures de cette fonction, et la variable indicatrice N_VE2 (cf. chapitre 4, § B-1-1,8).

NF	NSF	NOP	NE	NS	NCO	NVA	NT	CPXT	DATE	PROF	NVE	NVE2
F_1	1	0	4	3	0	14	6	1.000000	150	0	2	1
F_2	1	2	3	3	2	24	4	21.050465	151		0	0
F_3	2	0	3	3	0	25	6	11.870365	151	1	2	1
F_4	3	0	4	5	0	23	5	9.285402	153	3	0	0
F_5	2	3	4	4	2	44	7	48.808872	153	2	1	1
F_3	2	0	3	3	0	23	7	7.169925	153	1	1	1
F_1	1	1	4	3	1	19	5	8.108524	153	0	1	1
F_6	3	4	3	3	4	42	6	48.505379	156	4	0	0
F_7	0	13	2	2	2	42	1	86.214722	156	4	0	0
F_5	2	3	5	4	2	44	6	49.808872	157	2	0	0
F_8	2	0	3	2	0	13	4	0.000000	157	3	0	0
F_9	1	2	4	2	1	21	5	11.452241	157	2	0	0
F_{10}	3	0	2	2	3	20	5	4.000000	157	4	0	0
F_3	2	0	3	3	0	26	8	7.169925	157	1	0	0
F_1	1	1	4	3	1	19	5	8.108524	157	0	0	0
F_{11}	2	4	3	3	2	32	5	38.061203	157	3	0	0

Nous notons que le deuxième enregistrement ne contient pas d'indication sur la profondeur de la fonction F_2 dans l'arbre de décomposition de l'application : c'est parce que la fonction F_2 était une fonction temporaire, éliminée par la suite de la décomposition.

REFERENCES BIBLIOGRAPHIQUES

[ALB-82] ALBIN Jean-Luc, FERREOL Robert - "Collecte et analyse de mesures de logiciel" - Technique et Science Informatiques - Vol. 1, N° 4 - 1982.

[BER-68] BERGE Claude - "Principes de combinatoire" - Dunod - Paris - 1968.

[BER-83] BERGE Claude - "Graphes" - Gauthier-Villars - Collection Mu (B) - 3^{e} édition - 1983.

[BGD-86] BERGERAND Jean-Louis - "LUSTRE : un langage déclaratif pour le temps réel" - Thèse de Docteur-Ingénieur INPG - Grenoble - Janvier 1986.

[BGD-87] BERGERAND Jean-Louis - "Présentation générale d'un atelier logiciel : SAGA" - Merlin Gerin - Publication interne SES/CTI/LOG/SAGA - 38050 GRENOBLE Cedex - 1987.

[BRY-89] BERRY Gérard - "Real-time programming : special purpose or general purpose languages" - Proceedings of the IFIP 11th World Computer Congress "Information Processing 89" - G. X. Ritter editeur, pp. 11-17 - Septembre 1989.

[CHA-79] CHAPIN Ned - "A measure of software complexity" - National Computer Conference, pp. 995-1002 - 1979.

[FON-85] FONT Véronique - "Une approche de la fiabilité des logiciels : modèles classiques et modèle linéaire généralisé" - Thèse de 3^{e} cycle - Toulouse - Septembre 1985.

[GAU-88] GAUDOIN Olivier, SOLER Jean-Louis - "Analyse statistique d'un modèle de fiabilité des logiciels" - Rapport de Recherche RR-781-M - Laboratoire T.I.M.3, B.P. 53 X, 38041 GRENOBLE Cedex - Juin 1988.

[HAL-77] HALSTEAD Maurice H. - "Elements of software science" - The Computer Science Library - Operating and Programming Systems Series - 1977.

[HAR-81a] HARRISON Warren A. MAGEL Kenneth I. - "A complexity measure based on nesting level" - A.C.M. SIGPLAN NOTICES, Vol. 16, N° 3, pp. 63-74 - Mars 1981.

[HAR-81b] HARRISON Warren A. MAGEL Kenneth I. - "A topological analysis of the complexity of computer programs with less than three binary branches" - A.C.M. SIGPLAN NOTICES, Vol. 16, N° 4, pp. 51-63 - Avril 1981.

[JEL-72] JELINSKI Z., MORANDA P. - "Software Reliability Research" - Statistical Computer Performance Evaluation - W. Freiberger, éditeur - New-York, Academic Press, pp. 465-484 - 1972.

[LAP-84] LAPRIE Jean-Claude - "Dependability modeling and evaluation of software-and-hardware systems" - Informatik-Fachberichte - Fehlertolerirende Rechensysteme - Springer Verlag - pp. 203-215 - Septembre 1984.

[LAP-89] LAPRIE Jean-Claude - "Sûreté de fonctionnement des systèmes informatiques et tolérance aux fautes" - Techniques de l'ingénieur - A paraître - 1989.

[LIC-87] LI H. F., CHEUNG W. K. - "An empirical study of software metrics" - IEEE Transactions on Software Engineering, Vol. SE-13, N° 6, pp. 697-708 - Juin 1987.

[LIT-81] LITTLEWOOD B., VERRALL J.L. - "Likelihood function of a debugging model for computer software reliability" - IEEE Transactions on Reliability, Vol. R-30, N° 2, pp. 145-148 - Juin 1981.

[MAC-76] MAC CABE Thomas J. - "A complexity measure" - IEEE Transactions on Software Engineering, Vol. SE-2, N° 4, pp. 308-320 - Décembre 1976.

[MUS-75] MUSA John D. - "A theory of software reliability and its application" - IEEE Transactions on Software Engineering, Vol. SE-1, N° 3, pp. 312-327 - Septembre 1975.

[SAK-84] SAKAROVITCH Michel - "Optimisation combinatoire : programmation discrète" - Editions Hermann, collection "Enseignement des sciences", 32 - 1984.

[SCH-78] SCHICK George J., WOLVERTON Ray W. - "An analysis of competing software reliability models" - IEEE Transactions on Software Engineering, Vol. SE-4, N° 2, pp. 104-120 - Mars 1978.

[SHA-83] SHANTIKUMAR J. G. - "Software reliability models : a review" - Microelectr. Reliability, Vol. 23, N° 5, pp. 903-943 - 1983.

[SHE-83] SHEN Vincent Y., CONTE Samuel D., DUNSMORE H. E. - "Software science revisited : a critical analysis of the theory and its empirical support" - IEEE Transactions on Software Engineering, Vol. SE-9, N° 2, pp. 155-165 - Mars 1983.

[SOL-88] SOLER Jean-Louis - "Remarques sur la fiabilité des systèmes présentant des fautes de conception - Application à la fiabilité des logiciels" - Rapport Technique RT-36 - Laboratoire T.I.M.3, B.P. 53 X, 38041 GRENOBLE Cedex - Mai 1988.

[VAN-75] VAN EMDEN M.H. - "An analysis of complexity" - Mathematical Centre Tracts, 35 - Mathematisch Centrum, Amsterdam - 1975.

[WOO-79]WOODWARD Martin R., HENNELL Michael A., HEDLEY David - "A measure of control flow complexity in program text" - IEEE Transactions on Software Engineering, Vol. SE-5, N° 1, pp. 45-50 - Janvier 1979.

BIBLIOGRAPHIE

MODELES DE CROISSANCE DE FIABILITE

• ABDALLA Abdel-Ghaly A., CHAN P.Y., LITTLEWOOD Bev. - "Evaluation of competing software reliabilty predictions" - IEEE Transactions on Software Engineering, Vol. SE-12, N° 9, pp. 950-967 - Septembre 1986.

• CASPI Paul, PILAUD Eric - "Modèles d'évaluation de la sûreté de fonctionnement de systèmes redondants" - Proceedings of the 3rd International Conference on Reliability & Maintainability - Toulouse - Octobre 1982.

• CASPI Paul, KOUKA Edmond Félix - "Un test statistique de correction de systèmes fondé sur le modèle de fiabilité du logiciel de Jelinski-Moranda" - 4e colloque international sur la fiabilité et la maintenabilité, pp. 596-602 - 1984.

• KANOUN Karama, LAPRIE Jean-Claude - "Modelling software reliability and availability from development-validation up to operation" - Rapport de Recherche N° 85-041 - L.A.A.S., 7 avenue du Colonel Roche, 31077 TOULOUSE Cedex - Mars 1985.

• KANOUN Karama - "Validation de modèles stochastiques ; application aux modèles de croissance de fiabilité du logiciel" - Rapport de recherche N° 86-002 - L.A.A.S., Toulouse - Janvier 1986.

• KANOUN Karama - "Croissance de la sûreté de fonctionnement des logiciels. Caractérisation - Modélisation - Evaluation" - Thèse d'état - L.A.A.S., Toulouse - Septembre 1989.

• KOUKA Edmond Félix - "Les modèles de fiabilité du logiciel : application aux essais de validation d'un système informatique critique" - Thèse de Docteur-Ingénieur de l'Université - Grenoble - Mai 1985.

• LAPRIE Jean-Claude - "Vers une théorie de la fiabilité du X-iel" - Rapport interne LAAS-CNRS - Octobre 1987.

• LAPRIE Jean-Claude, COSTES Alain - "Dependability : a unifying concept for reliable computing" - Actes du 12^{e} symposium international "Fault-tolerant computing" (FTCS-12), Los Angeles - Juin 1982.

• LAPRIE Jean-Claude - "Towards a taxonomy of dependable computing" - Proc. 4th Jerusalem Conference on Information Technology, pp. 202-210 - Mai 1984.

• LITTLEWOOD B. - "Software reliability - Achievement and assessment" - Blackwell Scientific Publications, edited by B. Littlewood - 1987.

• RAMAMOORTHY C. V., BASTANI Farokh B. - "Software reliability - Status and perspectives" - IEEE Transactions on Software Engineering, Vol. SE-8, N° 4, pp. 354-371 - Juillet 1982.

• ROBINSON David G., DIETRICH Duane - "A new nonparametric growth model" - IEEE Transactions on Reliability, Vol. R-36, N° 4, pp. 411-418 - October 1987.

• SHOOMAN Martin L. - "Software reliability - A historical perspective" - IEEE Transactions on Reliability, Vol. R-33, N° 1, pp. 48-55 - Avril 1984.

ETUDES SUR LA COMPLEXITE

• BALL Michael O. - "Computational complexity of network reliability analysis : an overview" - IEEE Transactions on Reliability, Vol. R-35, N° 3, pp. 230-239 - Août 1986.

• BLUM Harold F. - "Complexity and organization" - Synthese, 15, pp. 115-121 - 1963.

• CORNACCHIO Joseph V. - "Maximum-entropy complexity measures" - Int. J. General Systems, Vol. 3, pp. 215-225 - 1977.

• CORNACCHIO Joseph V. - "System complexity - A bibliography" - Int. J. General Systems, Vol. 3, pp. 267-271 - 1977.

• DASTYCH Premysl - "A measure of the complexity of a subsystem" - Cybernetica, Vol. XVI, N° 1, pp. 58-72 - 1973.

• FERDINAND Arthur E. - "A theory of system complexity" - Int. J. General Systems, Vol. 1, pp. 19-33 - 1974.

• FLOOD Robert L., CARSON Ewart R. - "Dealing with complexity" - Plenum Press, New York - 1988.

• GEORGE Larry - "Tests for system complexity" - Int. J. General Systems, Vol. 3, pp. 253-258 - 1977.

• HARTMANIS J., HOPCROFT J. E. - "An overview of the theory of computational complexity" - Journal of the Association for Computing Machinery, Vol. 18, pp. 444-475 - Juillet 1971.

• HIGASHI Masahiko, KLIR George J. - "Measures of uncertainty and information based on possibility distributions" - Int. J. General Systems, Vol. 9, pp. 43-58 - 1982.

• KEMENY John G. - "Two measures of complexity" - The journal of philosophy, Vol. LII, N° 24, pp. 722-733 - Novembre 1955.

• KLIR George J. - "Complexity : some general observations" - Systems Research, Vol. 2, N° 2, pp. 131-140 - 1985.

• LOVELAND D. W. - "A variant of the Kolmogorov concept of complexity" - Information and control, 15, pp. 510-526 - 1969.

• MOWSHOWITZ Abbe - "Entropy and the complexity of graphs : I. An index of the relative complexity of a graph" - Bulletin of mathematical biophysics, Vol. 30, N° 1, pp. 175-204 - 1968.

• MOWSHOWITZ Abbe - "Entropy and the complexity of graphs : II. The information content of digraphs and infinite graphs" - Bulletin of mathematical biophysics, Vol. 30, N° 2, pp. 225-240 - 1968.

• RISSANEN Jorma - "Stochastic complexity and modeling" - The Annals of Statistics, Vol. 14, N° 3, pp. 1080-1100 - 1986.

• RISSANEN Jorma - "Stochastic complexity" - Journal of Royal Statistical Society, 49, N° 3, pp. 223-239 and 252-265 - 1987.

• ROSEN Barry K. - "Syntactic complexity" - Information and control, 24, pp. 305-335 - 1974.

• ROSEN Robert - "Complexity and errors in social dynamics" - Int. J. General Systems, Vol. 2, pp. 145-148 - 1975.

• WEAVER Warren - "Science and complexity" - American Scientist, Vol. 36, pp. 536-544 - 1948.

• WEIHRAUCH Klaus - "The computational complexity of program schemata" - Information and control, 24, pp. 305-335 - 1974.

• WILLIS David G. - "Computational complexity and probability constructions" - Journal of the Association for Computing Machinery, Vol. 17, N° 2, pp. 241-259 - Avril 1970.

• ZADEH Lofti A. - "Outline of a new approach to the analysis of complex systems and decision processes" - IEEE Transactions on Systems, Man, and Cybernetics, Vol. SMC-3, N° 1, pp. 28-44 - Janvier 1973.

COMPLEXITE DES LOGICIELS

• ALBRECHT Allan J., GAFFNEY JR. John E. - "Software function, source lines of code, and development effort prediction : a software science validation" - IEEE Transactions on Software Engineering, Vol. SE-9, N° 6, pp. 639-648 - November 1983.

• BAKER Albert L., ZWEBEN Stuart H. - "The use of software science in evaluating modularity concepts" - IEEE Transactions on Software Engineering, Vol. SE-5, N° 2, pp. 110-120 - Mars 1979.

• BAKER Albert L., ZWEBEN Stuart H. - "A comparison of measures of control flow complexity" - IEEE Transactions on Software Engineering, Vol. SE-6, N° 6, pp. 506-512 - Novembre 1980.

• BASILI Victor R., SELBY JR. Richard W., TSAI-YUN Phillips - "Metric analysis and data validation across Fortran projects" - IEEE Transactions on Software Engineering, Vol. SE-9, N° 6, pp. 652-663 - Novembre 1983.

• BASILI Victor R., HUTCHENS David H. - "An empirical study of a syntactic complexity family" - IEEE Transactions on Software Engineering, Vol. SE-9, N° 6, pp. 664-672 - Novembre 1983.

• BASILI Victor R., SELBY JR. Richard W. - "Calculation and use of an environment characteristic software metric set" - IEEE Proceedings of the International Conference on Software Engineering, pp. 386-391 - 1985.

• CHEN Edward T. - "Program complexity and programmer productivity" - IEEE Transactions on Software Engineering, Vol. SE-4, N° 3, pp. 187-194 - Mai 1978.

• CHEVALIER Marcel - "Complexité des logiciels de type Flots de Données" - Rapport Technique RT-46 - Laboratoire T.I.M.3, B.P. 53 X, 38041 GRENOBLE Cedex - Décembre 1988.

• CHEVALIER Marcel - "Fiabilité des logiciels Flots de Données" - Rencontres autour de la sûreté des systèmes électroniques - Merlin Gerin, Centre Paul-Louis Merlin - Juin 1988.

• CURTIS Bill, SHEPPARD Sylvia B., MILLIMAN Phil, BORST M. A., LOVE Tom - "Measuring the psychological complexity of software maintenance tasks with the Halstead and Mac Cabe metrics" - IEEE Transactions on Software Engineering, Vol. SE-5, N° 2, pp. 96-104 - Mars 1979.

• DUNSMORE Hubert E., GANNON John D. - "Experimental investigation of programming complexity" - Proceedings of the ACM-NBS 16th Annu. Tech. Symp. : System Software, pp. 117-125 - Mars 1979.

• FITZSIMMONS Ann, LOVE Tom - "A review and evaluation of software science" - A.C.M. Computing Surveys, Vol. 10, N° 1, pp. 3-18 - Mars 1978.

• GONG Huisheng, SCHMIDT Monika - "A complexity measure based on selection and nesting" - A.C.M. SIGMETRICS, Performance Evaluation Review, Vol. 13, N° 1, pp. 14-19 - Juin 1985.

• GORDON Ronald D. - "Measuring improvements in program clarity" - IEEE Transactions on Software Engineering, Vol. SE-5, N° 2, pp. 79-90 - Mars 1979.

• GORDON Ronald D. - "A qualitative justification for a measure of program clarity" - IEEE Transactions on Software Engineering, Vol. SE-5, N° 2, pp. 121-128 - Mars 1979.

• HARRISON Warren A., MAGEL Kenneth I. - "A complexity measure based on nesting level" - A.C.M. SIGPLAN NOTICES, Vol. 16, N° 3, pp. 63-74 - Mars 1981.

• HARRISON Warren A., MAGEL Kenneth I. - "A topological analysis of the complexity of computer programs with less than three binary branches" - A.C.M. SIGPLAN NOTICES, Vol. 16, N° 4, pp. 51-63 - Avril 1981.

• HELLERMAN Leo - "A measure of computational work" - IEEE Transactions on Computers, Vol. C-21, N° 5, pp. 439-446 - Mai 1972.

• ITAKURA Minoru, TAKAYANAGI Akio - "A model for estimating program size and its evaluation" - IEEE Proceedings of the International Conference on Software Engineering, pp. 104-109 - 1982.

• LASSEZ J.L., VAN DER KNIJFF D., SHEPHERD J. - "A critical examination of software science" - The Journal of Systems and Software, 2, pp. 105-112 - 1982.

• MOHANTY Siba N. - "Models and measurements for quality assessment of software" - A.C.M. Computing Surveys, Vol. 11, N° 3, pp. 251-275 - Septembre 1979.

• PEREZ Daniel - "Métrologie et analyse de données" - Thèse de 3^{e} cycle - Institut de Statistique des Universités de Paris - Octobre 1988.

• PHAM VAN Ngo - "Rôle des métriques de complexité d'un programme source dans l'assurance qualité logiciel" - 5^{e} colloque international de Fiabilité et de Maintenabilité, pp. 75-79 - Biarritz - Octobre 1986.

• RAMAMURTHY Bina, MELTON Austin - "A synthesis of software science measures and the cyclomatic number" - IEEE Transactions on Software Engineering, Vol. 14, N° 8, pp. 1116-1121 - Août 1988.

• SCHNEIDEWIND Norman F. - "Application of program graphs and complexity analysis to software development and testing" - IEEE Transactions on Reliability, Vol. R-28, N° 3, pp. 192-198 - Août 1979.

• SHATZ Sol M. - "Towards complexity metrics for ADA tasking" - IEEE Transactions on Software Engineering, Vol. 14, N° 8, pp. 1122-1127 - Août 1988.

• SHEN Vincent Y., YU Tze-jie,THEBAUT Stephen M., PAULSEN Lorry R. - "Identifying error-prone software : an empirical study" - IEEE Transactions on Software Engineering, Vol. SE-11, N° 4, pp. 317-324 - Avril 1985.

• SOI Inder M. - "Software complexity : an aid to software maintainability" - Microelectronics Reliability, Vol. 25, N° 2, pp. 223-228 - 1985.

• THIREAU Philippe - "Méthodologie d'analyse des effets des erreurs logiciel (AEEL) appliquée à l'étude d'un logiciel de haute sécurité" - 5e colloque international de Fiabilité et de Maintenabilité, pp. 111-117 - Biarritz - Octobre 1986.

• WOODFIELD Scott N. - "An experiment on unit increase in problem complexity" - IEEE Transactions on Software Engineering, Vol. SE-5, N° 2, pp. 76-79 - Mars 1979.

• ZOLNOWSKI Jean Cochrane, SIMMONS Dick B. - "Measuring complexity in structured vs unstructured cobol programs" - Computer & Industrial Engineering, Vol. 4, pp. 103-114 - 1980.

• ZUSE Horst, BOLLMANN Peter - "Software metrics : using measurement theory to describe the properties and scales of static software complexity metrics" - A.C.M. Sigplan Notices - Vol. 24, N° 8, pp. 23-33 - Août 1989.

FLOTS DE DONNEES

• ACKERMAN William B. - "Data flow languages" - National Computer Conference, pp. 1087-1095 - 1979.

• ASHCROFT E.A., WADGE W.W. - "Lucid, a nonprocedural language with iteration" - Communications of the A.C.M., Vol. 20, N° 7, pp. 519-526 - Juillet 1977.

• BERGERAND Jean-Louis, CASPI Paul, HALBWACHS Nicolas, PLAICE John A. - "Automatic control systems programming using a real-time declarative language" - IFAC/IFIP symposium SOCOCO, Graz - Mai 1986.

• CASPI Paul, PILAUD Daniel, HALBWACHS Nicolas, PLAICE John A. - "LUSTRE : a declarative language for programming synchronous systems" - 14th A.C.M. symposium on Principles of Programming Languages - Munich - Janvier 1987.

• KAVI M. Krishna, BUCKLES Bill P., BHAT U. Narayan - "A formal definition of data flow graph models" - IEEE Transactions on Computers, Vol. C-35, N° 11, pp. 940-947 - Novembre 1986.

• KAVI M. Krishna, BHAT U. Narayan - "Reliability analysis of computer systems using dataflow graph models" - IEEE Transactions on Reliability, Vol. R-35, N° 5, pp. 529-532 - Décembre 1986.

COMPLEMENTS TECHNIQUES

• BOUROCHE Jean-Marie, SAPORTA Gilbert - "L'analyse des données" - Presses universitaires de France - Collection "Que sais-je ?" - 3e édition - avril 1987.

• CAILLIEZ F., PAGES J.P. - "Introduction à l'analyse des données" - Société de Mathématiques Appliquées et de Sciences Humaines - Paris - 1976.

• PAGES Alain, GONDRAN Michel - "Fiabilité des systèmes" - Collection de la direction des Etudes et Recherches d'Electricité de France - Editions Eyrolles - 1980.

• PETERSON Wesley W., WELDON E. J. Jr - "Error-correcting codes" - M.I.T. Press, 2e édition - Octobre 1980.

• SAKAROVITCH Michel - "Optimisation combinatoire : graphes et programmation linéaire" - Editions Hermann, collection "Enseignement des sciences", 31 - 1984.

• WIRTH Niklaus - "Algorithms & Data Structures" - Prentice-Hall, New-Jersey - 1986.

TABLE DES ILLUSTRATIONS

AUTORISATION DE SOUTENANCE

DOCTORAT 3ème CYCLE, DOCTORAT INGENIEUR, DOCTORAT DE L'UNIVERSITE JOSEPH FOURIER - GRENOBLE 1

Vu les dispositions de l'Arrêté du **16 avril 1974,**

Vu les dispositions de l'Arrêté du **5 juillet 1984,**

Vu les rapports de M. Jean-Claude LAPRIE, Directeur de Recherches CNRS ;

M. Paul CASPI, Chargé de Recherches CNRS,

M. Marcel CHEVALIER est autorisé à présenter une thèse en vue de l'obtention du doctorat de l'Université Joseph Fourier - Grenoble 1, option Informatique.

Grenoble, le 22 NOV. 1989

Le président de l'Université Joseph Fourier - Grenoble 1

UNIVERSITE JOSEPH FOURIER GRENOBLE

A. NEMOZ