Rapport final

sur le projet

Techniques symboliques pour l'analyse statique et la vérification des systèmes

correspondant aux décisions d'aide 92 S 0605, 92 S 0606 et 92 S 0607 du 26 novembre 1992 visant conjointement

BULL, l'IMAG et le LIENS

au titre de l'appel d'offres "Informatique 92" du MRT

1 Introduction

L'utilisation de méthodes formelles pour concevoir les systèmes informatiques est une nécessité de plus en plus largement admise, aussi bien dans l'industrie du logiciel que dans celle du matériel. L'analyse sémantique et la vérification formelle sont deux disciplines voisines qui relèvent de ces méthodes formelles. Les équipes de recherche de Bull S.A. (Les Clayes-sous-Bois), de l'IMAG (Grenoble) et du LIENS (Paris) ont développé des compétences particulières dans chacun de ces domaines. Le projet correspondant à la décision d'aide 92 S 0605 du 26 novembre 1992 visait à mettre ces compétences en commun, dans le but d'opérer une véritable synergie tant des efforts de recherche que des applications industrielles.

Le projet, originellement prévu pour une durée de deux ans, a été prolongé à deux reprises, une première fois pour dix-huit mois à partir du terme initialement prévu, le 26 novembre 1994, une deuxième fois pour six mois à partir du second terme, le 26 mai 1996. La raison en était la suivante. Originellement, un échéancier des paiements par le ministère était prévu, échelonné par tranches de deux mois. Les équipes de recherche comptaient sur cet apport de fonds pour financer l'avancement des travaux demandés. Il apparut cependant que les fonds ne seraient versés qu'après le terme du projet. Eu égard, d'une part, aux délais nécessaires aux services du ministère pour effectuer les paiements aux laboratoires, et d'autre part, aux contraintes particulières de gestion des laboratoires publics (LIENS) ou d'économie mixte (IMAG), notamment l'impossibilité d'y effectuer des avances de trésorerie et donc la contrainte subie par ces laboratoires de ne pouvoir présenter leurs justificatifs de frais qu'après la date de fin du contrat, il était craint que ces laboratoires risquent de ne pas recevoir l'aide financière escomptée pour leurs travaux, pour cause de forclusion. C'est pourquoi nous avions demandé une prolongation du contrat.

En absence de financement de la part du ministère, les laboratoires ne disposaient donc pas des moyens nécessaires à engager les ressources initialement prévues pour accomplir les travaux initialement envisagés. Chaque laboratoire a donc dû revoir son organisation et ses moyens de financement. Les résultats obtenus sont malgré tout dans la ligne de ce qui était initialement prévu, et les engagements des trois équipes de recherche ont été tenus, sinon à la lettre, du moins dans leurs aspects fondamentaux : réalisations techniques, coopération, collaboration des équipes, et intégration des outils de chaque équipe.

Nous rappelons en section 2 les buts originaux du projet, ainsi que les motivations et les travaux qui étaient envisagés par les trois laboratoires de Bull, du LIENS et de l'IMAG; nous en profitons pour mentionner brièvement à l'occasion ce qui a effectivement été accompli, et ce qui a dû être revu. En section 3, nous décrivons rapidement comment la collaboration entre les équipes s'est installée lors du déroulement de l'étude. Puis, nous présentons en détail les travaux réalisés par chaque équipe, celle de l'IMAG en section 4, suivie de celle du LIENS en section 5, et enfin celle de Bull en section 6. Dans ces trois dernières parties, chaque section ou sous-section qui participe plus particulièrement d'un objectif de l'étude a reçu un titre de la forme :

4.1.1 Interface commune aux bibliothèques de BDD $\boxed{\textbf{Tâche 2.7.2}}$

signifiant par là que la section 4.1.1 rentre dans le cadre de la tâche rappelée en section 2.7.2; dans le cas présent, le travail de conception et de réalisation d'interfaces d'utilisation de modules de base.

Nous concluons en section 7.

2 Offre

2.1 Analyse sémantique et vérification formelle

L'analyse sémantique consiste à extraire des informations d'une description du système à un certain niveau de sa conception. Ces informations peuvent être utilisées pour s'assurer de la cohérence de la description et/ou pour optimiser la réalisation. En particulier, les techniques d'analyse sémantique sont susceptibles d'accroître substantiellement les capacités des compilateurs de langages de haut niveau, concernant la vérification statique de cohérence et l'optimisation du code produit. Elles sont également utilisées pour la parallélisation automatique des programmes.

La vérification formelle consiste à s'assurer formellement de la correction du système, en comparant généralement deux descriptions: La "spécification", qui décrit le comportement attendu, et la "réalisation", qui décrit le comportement du système effectivement implanté. La vérification formelle est un enjeu majeur dans des domaines critiques, où les erreurs de conception peuvent avoir des conséquences catastrophiques (temps réel, télécommunications, circuits,transactionnel ...).

2.2 Les approches classiques en analyse sémantique

Les méthodes d'analyse sémantique sont essentiellement prescriptives (contrôle de type), ce qui requiert une spécification partielle du programme (déclaration de type) ou analytiques (inférence de type [Mon91], analyse de flot de données, interprétation abstraite [CC77a],

[CC79] [CC92d]) pour lesquelles le programmeur ne fournit aucune information sur le programme, la seule spécification disponible étant la sémantique du programme. Dans les deux cas l'analyse est entièrement automatique, sans intervention du programmeur, ce qui en assure la fiabilité. Les méthodes d'analyse sémantique qui sont indépendantes des langages de programmation pour lesquels on les utilise sont essentiellement l'analyse de flot de données et l'interprétation abstraite. L'analyse de flot de données repose le plus souvent sur des arguments syntaxiques (comme les instances de définition et d'utilisation de variables) et se limite à des domaines finis (vecteurs de booléens le plus souvent). L'interprétation abstraite consiste à découvrir des informations sémantiques valides pour toutes les exécutions possibles du programme par approximation discrète des équations de point fixe définies par la sémantique (opérationnelle, dénotationnelle, axiomatique, etc.) du langage. La perte d'information est inévitable à cause des problèmes d'indécidabilité. De ce fait les principales difficultés de l'interprétation abstraite consistent à prendre en compte le contrôle d'exécution des programmes (procédures et fonctions d'ordre supérieur [Bou90], retour-arrière [CC92a], etc.) et à découvrir des algorithmes de codage et de manipulation de classes de propriétés des objets manipulés par les programmes. Par exemple la représentation approchée d'ensembles de valeurs numériques a été largement étudiée et appliquée avec succès :

1. intervalles d'entiers [CC76], [CC77a], c'est-à-dire des propriétés de la forme $\bigwedge_{i=1}^{n} a_i \leq x_i \leq b_i$ où les $a_i \leq b_i$ sont des constantes numériques découvertes par l'analyse et les x_i dénotent les valeurs numériques des variables;

2. variétés linéaires $\bigwedge_{i=1}^{n} a_i^1 x_i^1 + \cdots + a_i^{n_i} x_i^{n_i} = c_i$ où x_i^1, x_i^2, ?, $x_i^{n_i}$ sont les valeurs des variables entières du programmes et les constantes a_i^1, a_i^2, $\ldots$, $a_i^{n_i}$ et c_i sont découvertes par l'analyse [Ka76];

3. polyèdres convexes $\bigwedge_{i=1}^{n} a_i^1 x_i^1 + \cdots + a_i^{n_i} x_i^{n_i} \leq c_i$ [CH78];

4. classes de congruence telles que : x congru à p modulo q où p et q sont des constantes entières découvertes par l'analyse [Gra89] ou conjonctions de propriétés relationnelles du type : $f(x_1, ?x_2, ?\ldots, ?xn)$ est congru à p modulo q où f est une application affine à coefficients entiers, x_1, x_2, ?, x_n sont les valeurs des variables entières du programmes et p et q sont deux constantes entières découvertes par l'analyse [Gra91b], [Gra91a];

5. Congruences trapézoïdales c'est-à-dire systèmes d'équations de congruence sur les variables entières x_i du programme de la forme $\bigvee_{\delta \in [a,a']} \lambda_1 x_1 + \cdots + \lambda_n x_n \equiv \delta \pmod{m}$ avec $\lambda_1, \ldots, \lambda_n, a, a', m \in$ Z.

Ces représentations approchées d'ensembles d'objets numériques peuvent également être utilisées pour analyser des propriétés d'objets informatiques non numériques. Par exemple les synchronisations de programmes paralléles peuvent être analysées par comptage des rangs de leurs communications [Mer90]. Pour l'analyse des partages de structures de données [Deu92b], [Deu92a], la décomposition unitaire préfixe d'un langage rationnel introduite en théorie des languages par S. Eilenberg, permet d'approcher des langages et des relations rationnelles sur un monoïde libre, en se ramenant à la représentation d'un ensemble de vecteurs.

L'analyse sémantique peut également être utilisée en vérification, lorsque les propriétés découvertes par l'analyse sont celles que l'on veut vérifier. Il s'agit à l'évidence d'une méthode incomplète, toutes les propriétés d'un système n'étant pas dérivables automatiquement. En tout état de cause, l'utilisation de l'analyse sémantique pour la vérification n'a été que peu étudiée jusqu'alors. Les "modules de base" du calcul symbolique peuvent certainement être un apport de l'analyse sémantique au domaine de la vérification.

2.3 Les approches classiques en vérification formelle

En ce qui concerne la vérification formelle, deux grandes approches sont actuellement utilisées :

- Les méthodes générales, qui consistent à ramener la vérification à la preuve d'un théorème mathématique, et qui font appel à des systèmes de démonstration automatique. Les méthodes de ce type les plus connues sont basée sur les langages Z [Spivey 89, Suf85] , VDM [BJ 82] et son successeur RSL [RG 92] [BD 92]. Cette approche nécessite en général une interaction importante, l'usager guidant le démonstrateur en lui fournissant des axiomes, des tactiques de preuves, et des lemmes intermédiaires à démontrer. L'usager ne peut donc être qu'un "expert" en vérification. Cette nécessaire interaction et les performances des démonstrateurs rendent coûteuse l'application de ces méthodes à des systèmes de taille réelle.

- Les méthodes fondées sur des modèles finis. Complètement automatiques, ces méthodes s'appliquent aux systèmes assimilables à des machines d'états finis, soit intrinsèquement (protocoles de communication, circuits), soit du fait d'une abstraction (on ignore certains aspects du système, comme le comportement des variables numériques, et on se ramène de ce fait à un système fini). Dans ce cas, la vérification peut être effectuée par un examen exhaustif des états du système. Les systèmes Xesar [QS82], EMC [CES86] et AUTO [Ver86] sont des exemples d'outils fondés sur une telle énumération exhaustive. Il est clair que la principale limitation de ce type d'outils est la taille du modèle examiné (les meilleurs outils actuels traitent des modèles de l'ordre du million d'états).

2.4 De l'énumératif au symbolique : Les méthodes booléennes

Une amélioration sensible de l'approche fondée sur les modèles consiste, plutôt que d'énumérer explicitement les états du système, à manipuler globalement des ensembles d'états, équivalents du point de vue de la propriété à vérifier. Ces ensembles d'états sont représentés symboliquement par leurs formules caractéristiques. Cette approche symbolique peut donc être considérée comme intermédiaire entre l'approche énumérative et celle qui fait appel à la démonstration automatique. Le problème est évidemment la manipulation symbolique des formules. Dans le domaine des circuits, cette approche symbolique a permis un saut quantitatif important dans la puissance des outils de vérification, grâce à l'apport des méthodes récentes de codage et de décision des formules booléennes en termes de "graphes binaires de décision" [Bry86, CBM89a, CM90a, BCM+89]. Toujours limités aux systèmes finis, ces outils automatiques permettent d'effectuer des vérifications avec des performances qui ne sont plus liées au nombre d'états du système, mais plutôt à la complexité des formules manipulées.

2.5 Pour aller plus loin : Application à d'autres théories

Les succès remportés par les méthodes booléennes provoquent depuis quelques années un regain d'intérêt pour l'application pratique des méthodes symboliques. La manipulation symbolique d'ensembles d'états représentés par des formules permet d'envisager la vérification automatique de systèmes ayant un nombre infini d'états. Par exemple, une équipe de l'Université de Stanford a montré [ACD90] que la plupart des contraintes de délai utilisées dans la description des systèmes temps réel pouvaient être prises en compte dans un modèle fini, sans perte d'information, grâce à une représentation symbolique simple d'ensembles de vecteurs d'entiers. Cette extension des méthodes symboliques nécessite l'étude du codage et de la manipulation d'ensembles d'états de types variés. Les méthodes booléennes permettent de manipuler symboliquement des ensembles quelconques de vecteurs de booléens. Si la prise en compte générale d'ensembles quelconques de vecteurs numériques est certainement hors d'atteinte, on peut cependant obtenir des résultats en se restreignant à certaines classes d'ensembles: ainsi, en analyse sémantique, des algorithmes de codage et de manipulation de nombreuses classes d'ensembles numériques ont été étudiés et appliqués avec succès (intervalles d'entiers [CC77a], variétés linéaires [Ka76], polyèdres convexes [CH78], classes de congruence [Gra89]). Nous avions présenté la compétence acquise sur ces "modules de base" du calcul symbolique comme un apport de l'analyse sémantique au domaine de la vérification; cet apport a été exploité à bon escient depuis le début du projet.

2.6 Objectifs

Il n'était pas dans nos objectifs de définir un nouvel outil d'analyse et de vérification plus ou moins universel. En effet, un tel objectif nous semblait, et nous semble toujours hors d'atteinte, en raison de la diversité des domaines d'application, et des langages de description et de spécification qui leur sont propres. Par ailleurs, notre expérience nous a convaincu qu'il ne peut y avoir de "solution miracle" en matière d'analyse et de vérification: les diverses approches existantes sont plutôt complémentaires et incomparables. Sur certains exemples, les méthodes énumératives convergent plus vite que les méthodes symboliques, et inversement. Il en est ainsi de toute stratégie d'analyse ou de preuve, qui réussira mieux que d'autres dans certains domaines, sans qu'il soit possible de choisir a priori la stratégie la mieux adaptée à un problème donné. Il est donc souhaitable de disposer de toute une palette d'outils.

L'objectif de la présente étude était plutôt de mettre en commun et de permettre la coopération entre un certain nombre de modules et techniques de base, qui sont, à l'évidence, d'usage général. Ainsi, un module de calcul booléen symbolique est certainement le socle commun de nombreux outils, et a effectivement été utilisé dans divers buts par les différentes équipes. Cependant l'utilisation conjointe d'un tel module avec un module spécialisé dans la manipulation de certaines formules d'arithmétique peut poser problème. Apporter une réponse pratique à ce type de question était un des principaux buts de l'étude proposée. Là encore, les diverses équipes ont su mettre en commun les techniques qu'elles avaient développé et en tirer des algorithmes combinant plusieurs techniques dans le but de résoudre des problèmes pratiques.

Par ailleurs, il est clair qu'il existe une étroite liaison entre les méthodes de vérification symbolique et l'analyse sémantique. Les résultats approchés fournis par une analyse sémantique sont des indications précieuses pour la vérification : par exemple, une approximation supérieure de l'ensemble des états d'un système peut certainement être utilisée pour réduire la complexité de sa vérification. A l'inverse, l'analyse sémantique peut tirer parti des techniques de vérification développées pour des sous-ensembles décidables de formules. Une analyse sémantique guidée par les propriétés à vérifier ainsi que l'utilisation des résultats de l'analyse sémantique pour la preuve de programmes concurrents restent également à étudier. La "fertilisation croisée" de ces techniques était un volet, plus prospectif, de l'étude. Pour des questions de priorités, ces points n'ont pas été développés autant que nous l'aurions souhaité, mais certaines avancées ont cependant été faites dans ce domaine.

Rappelons pour finir que le but de ce projet était d'offrir un ensemble de techniques et d'outils d'analyse sémantiques de programmes en vue de vérifier leur bon fonctionnement d'une part et de faciliter leur maintenance d'autre part.

Ce travail s'insérait pour chaque partenaire dans un projet plus spécifique d'analyse ou de vérification de système auquel il fournit des outils et des techniques de base.

2.7 Programme détaillé

2.7.1 Modules de base de manipulation symbolique

La première tâche consistait, selon ce que nous avions écrit dans le rapport intermédiaire fourni en fin d'année 1993, en la mise en commun de modules logiciels (existants ou à concevoir) spécialisés dans la manipulation symbolique de certaines classes de formules: formules booléennes, intervalles d'entiers, relations linéaires entre variables numériques, congruences, formules particulières de la logique du premier ordre etc... Il était prévu que ces modules devaient implémenter les opérations de base (conjonction, disjonction, application de fonctions, ...) avec un souci d'efficacité maximale, aussi bien dans la représentation interne des formules que dans l'exécution des opérations. En guise de travail préparatoire, nous avions prévu de définir l'ensemble des opérations de base communes à chaque module, et de celles plus spécifiques à chacun. Patrick Cousot (LIENS) avait à cette occasion fourni une spécification des fonctions de base utilisées en interprétation abstraite qui devait servir de support à un système d'interfaçage hypothétique entre tous les modules fournis. Il s'est avéré qu'en pratique, un tel système générique d'interfaçage représentait une utopie plutôt qu'un véritable moyen de faciliter la mise en commun des modules. Nous avons adopté une solution plus pratique quoique scientifiquement moins esthétique: la mise en commun des modules s'est tout simplement faite via les outils disponibles sur Internet (ftp, mail, Web), et chacun a utilisé les modules fournis par les autres équipes en les adaptant selon ses besoins propres, et en profitant de l'expertise de ses partenaires.

Les modules de base :

- Module de traitement des graphes de décision binaires (BDD), module de calcul booléen symbolique;
- Modules de manipulation d'intervalles, de polyèdres convexes, de congruences ...
- Modules de propagation d'intervalles, de polyèdres, de congruences, ...

Ces modules, quoique spécifiques, ont été nécessaires à toutes les équipes, et utilisés dans diverses applications par chacun des partenaires.

Des modules d'usage général s'appuyant sur ces derniers devaient aussi être fournis :

Modules auxiliaires :

- Module de preuve.
- Module d'analyse interprocédurale.
- Module de détection de réductions généralisées.

Ces modules ont été réalisés; cependant, il semble qu'en général ces modules aient été d'un intérêt uniquement pour les équipes qui les avaient conçus, et ils n'ont donc pas eu le succès des modules de base.

Apport de chaque partenaire

IMAG :

- Module de manipulation de "zones", cas particulier de systèmes de contraintes line'aires à coefficients dans $\{-1, 0, 1\}$, utilise's pour l'analyse des contraintes temps réel [ACD90].
- Module d'analyse de graphes temporisés, utilisant le module précédent.
- Module de construction de graphe d'états réduits, fondé sur sur les modules de base et sur l'algorithme original décrit dans [BFH$^+$92].

LIENS : La technique de l'interprétation abstraite développée par Cousot constitue l'apport théorique du LIENS. Un grand nombre de modules de manipulation symbolique ont été étudiés dans l'équipe (intervalles d'entiers, variétés linéaires, polyèdres convexes, classes de congruence, congruences trapézoïdales). Certains de ces modules ont été adaptés dans les autres équipes, et de nouveaux modules ont été réalisés à partir des techniques développées par les autres partenaires.

BULL :

- Analyse interprocédurale de flots de données dont l'objectif est de calculer les effets de chaque instruction (simple ou composée) du programme. Cette analyse est basée sur une évaluation symbolique du programme couplée avec une technique de détections de réductions généralisées.

- Module de preuve pour la logique du premier ordre dont la principale originalité est d'utiliser les techniques des BDD, de manière à vérifier beaucoup plus efficacement si la formule propositionnelle obtenue par substitution des prédicats par des symboles propositionnels est valide [BOL 91], [BIL 91].
- Les graphes de décision binaires (BDD)

2.7.2 Interface d'utilisation des modules

Un deuxième volet visait à définir et implanter une interface commune d'utilisation des modules de base. Un format général de représentation externe des formules et des opérations devait être défini dans cette tâche. Le caractère utopique de ce deuxième volet (voir plus haut) a forcé les différentes équipes à adopter une attitude plus pratique, consistant à interfacer les différents modules au cas par cas.

2.7.3 Intégration aux outils existants

Chaque partenaire devait intégrer alors le résultat des deux tâches précédentes à ses propres outils, et en étudier les extensions rendues possibles.

IMAG : L'utilisation des techniques symboliques permet un saut qualitatif important dans les outils de vérification développés à l'IMAG: les systèmes XESAR, CÆSAR et LESAR, respectivement dédiés à la vérification de programmes écrits dans les langages Estelle, Lotos et Lustre. En effet, ces systèmes étaient limités à la vérification de programmes d'états finis de taille moyenne (de l'ordre du million d'états). Les techniques booléennes symboliques permettaient souvent de dépasser largement cette taille. La manipulation symbolique de variables entières permet de traiter des programmes ayant un nombre infini d'états. Enfin, l'IMAG a fourni un outil d'analyse des systèmes temps réel fondé sur la manipulation symbolique de "zones", à partir d'un graphe temporisé.

BULL : L'utilisation des outils offerts par ce projet devait ouvrir la voie à des techniques d'analyse plus puissantes. Cela devait être effectué en intégrant la propagation d'intervalles de valeurs ou de contraintes linéaires dans l'évaluateur symbolique. En fait, ce sont les techniques de propagation de polyèdres, avec les techniques de BDD et quelques autres qui ont été utilisées.

2.7.4 Combinaison des approches

L'étude devait tenter de faire converger les deux approches: interprétation abstraite et analyse des systèmes finis. Rappelons que les résultats approchés fournis par une analyse sémantique sont des indications précieuses pour la vérification et que réciproquement l'analyse sémantique peut tirer parti des techniques de vérification développées pour des sous-ensembles décidables de formules. La "fertilisation croisée" de ces techniques était un volet prospectif de l'étude, qui a été initié lors du projet; cet effort demande à être poursuivi.

2.7.5 Conséquences attendues

Un objectif général de cette étude était de favoriser la coopération de trois équipes ayant des compétences complémentaires sur le sujet, toujours très actuel, de l'analyse automatique des systèmes. Plus concrètement, les conséquences attendues de cette coopération concernaient d'une part le développement de nouveaux outils d'analyse et de vérification, et d'autre part l'émergence de résultats plus fondamentaux sur les techniques d'analyse.

Outils: L'intégration des résultats de l'étude aux outils existants, en matière de vérification, de compilation et de parallélisation, a pour but d'augmenter substantiellement leur puissance et donc leur applicabilité. Il était envisagé que de nouveaux outils puissent même voir le jour à la suite de cette étude. Ceci a effectivement été le cas; on citera notamment l'outil POLKA chez VERIMAG, ou le projet Aspic chez Bull.

Résultats fondamentaux: Il était envisagé que la combinaison des deux approches citées devait résulter en de nouvelles technique d'analyse, plus puissantes et performantes. Ceci a été le cas dans les trois équipes concernées, comme on le verra en détail dans les sections suivantes.

3 Organisation des travaux

Les rencontres entre les différents partenaires en présence ont été fréquentes pendant l'année 1993. Deux réunions entre les partenaires ont été organisées, et les séminaires "Sémantique et Interprétation Abstraite" (SIA) organisés par Patrick Cousot au LIENS ont été de nombreuses occasions de mettre en présence les acteurs du projet. Ces réunions ont essentiellement permis de structurer le projet, de décider de la conduite des activités des différents partenaires au sein du projet, et bien sûr de faire se connaître les membres des différentes équipes. Par la suite, ces réunions se sont faites moins fréquentes, et les échanges se sont davantage effectués via Internet ou par téléphone; le séminaire SIA est cependant resté une occasion régulière de rencontres.

Une première réunion s'est tenue à Grenoble, dans les locaux de l'IMAG, les 14 et 15 janvier 1993. Le but de cette réunion était de faire un tour d'horizon des techniques et méthodes proposées par chaque centre de recherche, ainsi que de faire prendre conscience aux différents acteurs du projet des similarités et des différences dans les besoins en matière de vérification formelle et d'analyse sémantique; et ensuite, de définir un plan d'action et un échéancier des différentes contributions des trois centres. La deuxième réunion, du 17 juin 1993 à Paris, confirmait ces prévisions et faisait le point de l'avancement des travaux.

La première réunion avait identifié quatre points fondamentaux pour l'activité des différentes parties dans le projet:

- mise à disposition de modules de manipulation symbolique,
- création d'interfaces d'utilisation des modules,
- intégration des modules aux outils existants,

- combinaison des approches.

Sur le premier point, VERIMAG (faisant partie de l'IMAG) a fourni ses deux modules de manipulation de polyèdres généraux, et de polyèdres restreints, en fin 1993. Bull a livré sa bibliothèque des BDD en juin; son module de preuve en logique du premier ordre était prêt en novembre 1993. Le LIENS a fourni les bibliothèques d'analyse d'analyse de nécessité (strictness) de Laurent Mauborgne, et d'analyse de programmes parallèles de Régis Cridlig.

Sur le deuxime point, il avait été décidé que la création d'interfaces d'utilisation des modules devait être conditionnée par la définition d'une syntaxe abstraite commune, c'est-à-dire d'un langage de communication entre un outil et un module. Le LIENS a fourni cette syntaxe abstraite en juillet 1993, qui reprenait un noyau d'opérations de base commun à toutes les approches, avec des conventions d'extension pour les opérations spécifiques à chacun. La ressemblance apparemment très forte entre les fonctionnalités de base demandées par tous (par exemple: un treillis d'interprétation abstraite, des fonctions de borne supérieure, borne inférieure, d'élargissement, etc.) est cependant trompeuse. Patrick Cousot a émis dès 1993 des doutes quant à l'utilisabilité d'une telle interface commune. Il s'est avéré ensuite qu'il serait illusoire de vouloir contraindre les outils des différentes équipes, qui répondent à des exigences variées, à rentrer dans le même corset. Une approche plus souple était donc exigée, et l'interfaçage entre les divers modules s'est le plus souvent faite au cas par cas. Les travaux de Régis Cridlig au LIENS ont cependant montré que si l'on se limitait à certains domaines d'interprétation abstraite — qui ne couvrent cependant pas le spectre entier des outils utilisés dans l'étude — il était possible de définir une telle interface standard (voir plus loin).

Les autres points, intégration aux outils existants et combinaison des approches, étaient remis à l'année 1994, le temps que les outils de base soient fournis et que les modalités d'utilisation de chaque module soient exposées à chacun. Il est plus difficile de donner une décomposition détaillée des activités dans ce domaine. Essentiellement, chaque partenaire a étendu son expertise en utilisant les outils des autres participants au projet. Ainsi les BDD et les bibliothèques de calculs sur les polyèdres convexes ont-ils été utilisés par les trois participants, par exemple.

4 Travaux effectués à l'IMAG

4.1 Méthodes Booléennes

4.1.1 Interface commune aux bibliothèques de BDD — Tâche 2.7.2

De nombreuses bibliothèques de BDD sont maintenant disponibles, et leur évolution est permanente. Pour faciliter les changements de bibliothèques, et leur comparaison, nous avons développé une interface standard permettant de rendre transparente au programme utilisateur la bibliothèque de BDD utilisée. Ecrite en C++, cette interface offre les opérations usuelles, indépendamment de la bibliothèque. Elle a été connectée à la bibliothèque développée à l'IMAG, et à celle de l'outil TiGeR [CMT93].

4.1.2 Optimisation des méthodes de vérification booléennes $\boxed{\textbf{Tâche 2.7.3}}$

La "redécouverte" récente [Bry86] des graphes binaires de décision (ou BDD [Ake78]) a permis de faire de grands progrès dans le domaine de la vérification formelle de machines finies [TSL+90, HD93, CBM89b]. Ce type de représentation permet en effet d'implémenter efficacement des opérations symboliques complexes (union, intersection, quantification etc). En particulier, le parcours exhaustif de l'ensemble des états d'une machine finie n'est plus intrinsèquement limité par sa taille : une formule symbolique simple (i.e. un petit BDD) peut éventuellement coder un énorme ensemble d'états. Ce type de méthodes symbolique à été implémenté dans l'outil LESAR [Rat92, HLR92] de l'IMAG, dédié à la vérification de propriétés de programmes Lustre. Cette section présente les optimisations apportées récemment aux méthodes symboliques utilisées dans cet outil.

Rappels Nous nous plaçons à un niveau d'abstraction où le système à vérifier est codé par une machine d'état finis. Plus précisément, cette machine est un automate représenté par un système d'équations booléennes. Ce système est un quadruplet (E, S, T, s_0) où :

- $E = \{e_1, \ldots, e_{|E|}\}$ est l'ensemble des variables libres (i.e. les entrées) de la machine.
- $S = \{s_1, \ldots, s_{|S|}\}$ est l'ensemble des variables d'états (i.e. les mémoires) de la machine.
- $T = \{t_1, \ldots, t_{|S|}\}$ est l'ensemble des fonctions de transitions ; chaque t_i est une fonction booléenne qui définit, en fonction des valeurs courantes des variables d'état et des entrées, la valeur suivante de la $i$ème variable d'état.
- s_0 est l'état initial de la machine, c'est-à-dire un vecteur de $|S|$ valeurs booléennes qui définit la valeur initiale de chaque variable d'état.

Un état de la machine est donc une configuration particulière des variables d'état. L'état $s' \in 2^{|S|}$ est un successeur de l'état s, si il existe un vecteur de valeur d'entrées $e \in 2^{|E|}$ tel que :

$$s' = [t_0(s, e), \ldots , t_{|S|}(s, e)]$$

L'ensemble des successeurs d'un état s est noté $\mathsf{post}(s)$; cette notion est étendue aux ensembles d'états : soit σ un ensemble d'état, on note $\mathsf{Post}(\sigma)$ l'union des $\mathsf{post}(s)$ pour chaque état s de σ. Parmi les $2^{|S|}$ états potentiels, seuls nous intéressent les états accessibles, c'est-à-dire ceux qu'on peut atteindre depuis l'état initial en appliquant pas à pas la fonction post. Cet ensemble, que nous notons $\mathcal{A}$, est clairement un plus petit point fixe, qui peut être calculé comme la limite d'une suite A, avec $A_0 = \{s_0\}$ et $A_{i+1} = A_i \cup \mathsf{Post}(A_i)$.

Nous supposons maintenant que la propriété à vérifier est une propriété de sûreté, qui s'exprime dans notre modèle par l'invariance d'une fonction booléenne $I(S, E)$, dépendant à la fois de l'état de la machine et de la valeur de ses entrées. Cette fonction, que nous appelons simplement l'invariant, permet de déduire l'ensemble des "mauvais états", c'est-à-dire les états pour lesquels il existe une valeur des entrées qui rend l'invariant faux :

$$\mathcal{B} = \{s | \exists e \quad \neg I(s, e)\}$$

Le but de la vérification formelle est de prouver qu'aucun état accessibles n'appartient à $\mathcal{B}$, c'est-à-dire que :

$$\mathcal{A} \cap \mathcal{B} = \emptyset$$

Méthode symbolique Dans les méthodes symboliques, les ensembles d'états sont représentés par des formules booléennes (i.e. leurs fonctions caractéristiques). D'un point de vue technique, ces formules booléennes sont codées par des graphes binaires de décision (BDD), qui permettent une implémentation efficace des opérations ensemblistes (union, intersection, quantification etc). En particulier, si chaque fonction de transition t_i est elle-même codée par un BDD, on dispose d'un algorithme efficace pour le calcul de la fonction Post. Grâce à ces techniques, l'algorithme de calcul des états accessibles peut être directement déduit de la définition de $\mathcal{A}$. Voici, en pseudo-C, le schéma général de cet algorithme :

```
bdd A = Empty;
bdd NA = initial;
while ( A != NA ) {
    A = NA ;
    NA = Union(A, Post(A));
}
```

Dans cet algorithme, initial est le codage en BDD du singleton contenant l'état initial ; Empty est la constante qui code l'ensemble vide (i.e. la fonction identiquement fausse). Les opérations d'affectation et de comparaison entre BDD ont un coût constant (négligeable), et l'opération Union a un coût polynomial (faible) par rapport à la taille de ses opérandes. Toute la complexité de cet algorithme réside donc dans le calcul de la fonction Post.

Utilisation d'un opérateur de restriction Une optimisation importante consiste à réduire la taille de l'argument du Post. Par exemple, à chaque itération, seul les états vraiment nouveaux (i.e. qui appartiennent à NA mais pas à A) doivent être traités. Le cœur de la boucle peut donc être remplacé par :

```
X = Intersection(NA, Complement(A));
A = NA;
NA = Union(A, Post(X));
```

Il est cependant à noter que la taille d'un BDD n'a aucun rapport avec le cardinal de l'ensemble qu'il code : il est possible que le BDD de Intersection(NA, Complement(A)) soit plus complexe que celui de A, bien qu'il code un sous-ensemble de ce dernier. Pour réduire la taille des BDD, il faut utiliser un opérateur particulier, dit de restriction : soient f et g deux BDD, $h = \mathsf{Restrict}(f, g)$ est un BDD équivalent à f quand g est vraie, mais de taille réduite. Avec cet opérateur, le cœur de la boucle devient :

```
X = Restrict(NA, Complement(A));
A = NA;
NA = Union(A, Post(X));
```

Intuitivement, X contient tous les états qui n'ont pas encore été traités, plus quelques états déjà traités, choisis de manière à ce que le BDD de X soit "plus simple" que celui de NA.

Utilisation de variables intermédiaires Les programmes sont en général initialement décrits par des expressions booléennes classiques, construites avec des opérateurs logiques (et, ou, non etc). Pour pouvoir appliquer les algorithmes précédents, il faut donc tout d'abord construire les BDD correspondant à ces expressions booléennes. Cette phase est particulièrement critique, car elle a un coût exponentiel par rapport à la taille du programme initial.

Pour faire face à ce problème, nous avons implémenté une solution qui ne nécessite que la construction de "petits" BDD.

L'idée est d'identifier dans la description initiale un ensemble de variables intermédiaires, en recherchant par exemple des sous-expressions partagées[1]. Soit L cet ensemble de variables, chaque variable $l_i \in L$ est décrite par une expression booléenne qui dépend des entrées, des variables d'état et d'autres variables intermédiaires. De même, chaque fonction de transition peut être interprétée comme une fonction booléenne dont les arguments sont dans $L \cup E \cup S$. Avec cette solution, on a plus de variables, mais les expressions sont plus simples, et les BDD résultant sont en général "petits".

Bien évidement, les BDD où apparaissent des variables intermédiaires ne sont pas directement exploitables dans l'algorithme de calcul des états accessibles : le calcul du Post nécessite un ensemble de fonctions de transition qui ne dépendent que des entrées et des variables d'état. Soit X l'ensemble d'états dont on veut calculer le Post, l'idée est de construire, en utilisant les techniques de restriction, une version simplifiée des fonctions de transition valable pour ce X particulier : soient $l_i : L \times E \times S \to \mathbb{B}$ les BDD des variables intermédiaires, et $t_i : L \times E \times S \to \mathbb{B}$ ceux des variables d'état. On commence par construire les familles de BDD simplifiés $l'_i = \mathsf{Restrict}(l_i, X)$ et $t'_i = \mathsf{Restrict}(t_i, X)$; puis on construit un vecteur de fonctions de transition exploitable $(t''_1, \ldots, t''_{|S|})$ en substituant récursivement dans les t'_i les variables intermédiaires par leurs définitions simplifiées (les l'_i).

Cette solution, qui nécessite de recalculer les fonctions de transitions à chaque itération peut sembler coûteuse, mais elle offre cependant une solution intéressante au traitement de programmes pour lesquels la construction d'un ensemble "global" de fonctions de transition n'est pas possible.

4.1.3 Utilisation d'approximations $\boxed{\textbf{Tâche 2.7.4}}$

Dans [HR96], nous avons étudié l'usage d'informations approchées, fournies par exemple par des techniques d'interprétation abstraite (voir en particulier, l'opérateur d'élargissement sur les BDD défini au LIENS [Mau94]; voir section 5.2), pour la vérification exacte, notamment par des méthodes boléennes fondées sur les BDD.

Par exemple considérons le cas simple où l'on veut vérifier une propriété invariante ([HR96] généralise à la vérification de formules de logique temporelle arborescente), c'est-à-

[1] il est en fait assez courant que de telles variables intermédiaires fassent déjà partie de la description : ce sont par exemple les variables locales d'un programme Lustre

dire qu'un ensemble donné d'états $\mathcal{P}$ contient tous les états qu'un programme peut atteindre. Deux approches peuvent être appliquées:

- La méthode *en arrière* consiste à construire le plus grand invariant inductif[2] $\mathcal{I}$ inclus dans $\mathcal{P}$, en vérifiant qu'il contient l'ensemble *init* des états initiaux du programme.
- La méthode *en avant* consiste à construire l'ensemble $\mathcal{A}$ des états accessibles, en vérifiant qu'il est inclus dans $\mathcal{P}$.

$\mathcal{A}$ et $\mathcal{I}$ sont tous deux calculés comme des points fixes, et peuvent être approchés par des méthodes d'interprétation abstraite. Notons $\lfloor X \rfloor$ et $\lceil X \rceil$, respectivement, une approximation inférieure et supérieure d'un ensemble X. Il est clair que la connaissance d'approximations de $\mathcal{A}$ et $\mathcal{I}$ peut fournir une méthode de vérification partielle : si $\lceil \mathcal{A} \rceil \subseteq \mathcal{P}$, ou si $init \subseteq \lfloor \mathcal{I} \rfloor$, la vérification est achevée. Inversement, si $\lfloor \mathcal{A} \rfloor \not\subseteq \mathcal{P}$, ou si $init \not\subseteq \lceil \mathcal{I} \rceil$, la propriété n'est pas satisfaite. Ces méthodes partielles devraient être tentées avant toute autre vérification. Cependant, nous nous sommes plutôt intéressés à *combiner* les méthodes d'approximation avec la vérification exhaustive.

Méthode en arrière : Dans cette approche, on calcule $\mathcal{I}$ comme limite d'un séquence décroissante :

$$\mathcal{I}_0 = \mathcal{P}, \quad \mathcal{I}_{n+1} = \mathcal{I}_n \setminus \mathsf{Pre}(\neg \mathcal{I}_n)$$

Il est clair que, pour prouver que $init \subseteq \mathcal{I}$, il suffit de prouver que $init \subseteq \mathcal{A} \cap \mathcal{I}$. Donc, une approximation supérieure $\lceil \mathcal{A} \rceil$ de $\mathcal{A}$ peut être utilisée de diverses manières au cours d'une vérification en arrière :

1. Au lieu de calculer le plus grand invariant inductif inclus dans $\mathcal{P}$, on peut calculer le plus grand invariant inductif inclus dans $\mathcal{P} \cap \lceil \mathcal{A} \rceil$. Cependant, il n'est pas évident, dans le cas des méthodes à base de BDD, que cette technique apporte une optimisation : elle est suscptible de hâter la convergence des itérations, mais les BDD construits peuvent être plus gros que dans la méthode originale.
2. On peut utiliser la "restriction" (voir §4.1.2) par $\lceil \mathcal{A} \rceil$ pour simplifier les BDD obtenus à chaque itération : chaque terme $\mathcal{I}_n$ peut être remplacé par n'importe quel ensemble X_n tel que

   $$\mathcal{I}_n \cap \lceil \mathcal{A} \rceil = X_n \cap \lceil \mathcal{A} \rceil$$

   Ce type de simplification conduit en général à des optimisations importantes.
3. On en conclut un meilleur critère d'arrêt des itérations : dès que $\mathcal{I}_{n+1} \cap \lceil \mathcal{A} \rceil = \mathcal{I}_n \cap \lceil \mathcal{A} \rceil$, le calcul peut être arrêté, parce que $\mathcal{A} \cap \mathcal{I}_n = \mathcal{A} \cap \mathcal{I}$.

[2] Rappelons qu'un ensemble X est un invariant inductif si et seulement si $\mathsf{Post}(X) \subseteq X$, ou, de manière équivalente, si $X \subseteq \mathsf{Pre}(X)$.

Méthode en avant : On calcule l'ensemble $\mathcal{A}$ comme limite d'une suite croissante

$$\mathcal{A}_0 = init, \quad \mathcal{A}_{n+1} = \mathcal{A}_n \cup \mathsf{Post}(\mathcal{A}_n)$$

On n'est alors intéressé qu'au calcul des états accessibles qui peuvent mener à la violation de la propriété $\mathcal{P}$. On peut donc utiliser une approximation supérieure de l'ensemble $\mathcal{V}$ des états susceptibles de mener à l'extérieur de $\mathcal{P}$. De manière analogue, $\lceil\mathcal{V}\rceil$ peut être utilisé pour simplifier le BDD de chaque terme $\mathcal{A}_n$, et la stabilisation de $\mathcal{A} \cap \lceil\mathcal{V}\rceil$ est un critère de terminaison.

4.2 Méthodes numériques : L'analyseur POLKA

L'outil Polka consiste en trois composants que nous décrivons en détail dans la suite :

- Une bibliothèque d'opérations de manipulation de polyèdres convexes (systèmes d'équations et d'inéquations linéaires).
- Un "calculateur de bureau" permettant d'activer interactivement les opérations de la bibliothèque.
- Un outil d'analyse et de vérification d'automates hybrides linéaires, fondé sur une interprétation abstraite en termes de polyèdres convexes.

Polka fait l'objet d'une distribution publique par Internet, et est utilisé dans de nombreux sites, en particulier:

- En France: aux laboratoires d'automatique de Grenoble et de Nantes, à l'Inria (Rhône-Alpes et Rocquencourt), au CERT (Toulouse) et au CEN (Saclay).
- A l'étranger: aux universités de Cornell, Berkeley et Stanford (USA), au Technion Institute (Israel), au Tata Institute (Inde), et à l'Ecole Polytechnique Fédérale de Lausanne (Suisse).

4.2.1 La bibliothèque d'opérations sur les polyèdres — Tâche 2.7.1

La bibliothèque est écrite en C++. Elle offre un ensemble d'opérations sur les polyèdres convexes, définis soit par un système d'équations et d'inéquations linéaires, soit par un "système générateur" qui est l'ensemble des sommets et des rayons (sommets à l'infini) du polyèdre (cf. Fig 1). On se place dans l'espace rationnel $\mathbb{Q}^{\mathsf{K}}$, n étant le nombre de variables du problème. Tous les calculs sont effectués en nombres rationnels, pour éviter les erreurs d'arrondi.

L'opération clé, qui permet de réaliser simplement toutes les autres, est la conversion d'une représentation dans l'autre. Cette conversion permet également, et c'est fondamental, de minimiser les représentations (élimination de contraintes redondantes, ou de sommets ou rayons non-extrémaux). Nous utilisons pour cela l'optimisation proposée par Hervé Leverge [Lev92] de l'algorithme de Chernikova [Che68]. Compte tenu de l'importance de cet algorithme, nous en décrivons maintenant les grandes lignes.

4.2.2 L'algorithme de Chernikova-Leverge

Nous présentons succinctement l'algorithme dans le cas de la recherche du système générateur d'un polyèdre donné par son système de contraintes. La conversion inverse s'obtient par dualité. L'algorithme procède par adjonctions successives de contraintes.

Initialement, on considère le polyèdre sans aucune contrainte, i.e., l'espace $\mathbb{Q}^{\kappa}$ tout entier, dont un système générateur minimal est donné par le seul sommet origine $\vec{0}$ et par l'ensemble des vecteurs de base et de leurs opposés comme rayons.

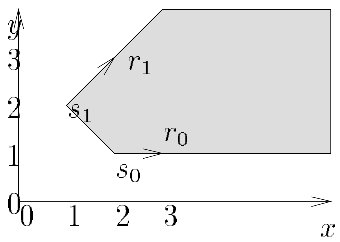
Pour ajouter une contrainte à un polyèdre dont on connaît déjà un système générateur minimal, on commence par partitionner les éléments générateurs selon qu'ils satisfont la nouvelle contrainte ou non. Tous ceux qui la satisfont sont générateurs du nouveau polyèdre. Les autres générateurs sont obtenus par combinaison linéaire d'un élément satisfaisant la contrainte avec un élément ne la satisfaisant pas. Le problème est que ces nouveaux éléments ne sont pas nécessairement extrémaux.

Considérons l'exemple de la Fig. 2 : l'ensemble $\{s_1, s_2, s_3, s_4\}$ des sommets du polyèdre initial est partitionné en $\{s_1, s_2\}$, qui satisfont la nouvelle inéquation, et $\{s_3, s_4\}$ qui ne la satisfont pas. Par combinaisons linéaires de ces sommets, on obtient les quatre nouveaux points $\{s_{1,3}, s_{1,4}, s_{2,3}, s_{2,4}\}$ sur la contrainte (où $s_{i,j}$ est la combinaison de s_i et s_j), dont seuls $s_{1,3}$ et $s_{2,4}$ sont réellement des sommets du résultat.

L'optimisation proposée par [Lev92] consiste à remarquer qu'un point (resp., vecteur) est un vrai sommet (resp., rayon) s'il sature au moins $n - p$ (resp., $n - p - 1$) contraintes, où p est la dimension du plus grand sous-espace contenu dans le polyèdre. Cette observation conduit à un critère simple pour éliminer les générateurs inutiles.

Les opérations de la bibliothèque On dispose ainsi d'une méthode efficace pour calculer les deux représentations d'un polyèdre, et pour maintenir ces représentations minimales. Les opérations ensemblistes utiles sont alors faciles à réaliser en utilisant l'une ou l'autre représentation :

Intersection : L'intersection de deux polyèdres est calculée par conjonction de leurs systèmes de contraintes.



$$P = \left\{ (x,y) \mid \begin{pmatrix} y & \geq & 1 \\ x+y & \geq & 3 \\ -x+y & \leq & 1 \end{pmatrix} \right\}$$

$$S = \left\{ s_0 \begin{pmatrix} 2 \\ 1 \end{pmatrix}, s_1 \begin{pmatrix} 1 \\ 2 \end{pmatrix} \right\} \quad R = \left\{ r_0 \begin{pmatrix} 1 \\ 0 \end{pmatrix}, r_1 \begin{pmatrix} 1 \\ 1 \end{pmatrix} \right\}$$

Figure 1: Les deux représentations d'un polyèdre convexe

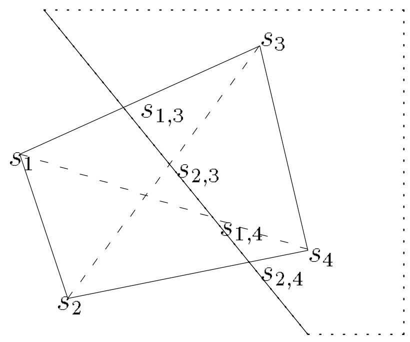


Figure 2: Adjonction d'une inéquation à un polyèdre

Enveloppe convexe : L'enveloppe convexe de deux polyèdres (plus petit polyèdre convexe les contenant, opération utilisée comme approximation supérieure de l'union) est obtenue en faisant l'union de leurs ensembles de sommets, et l'union de leurs ensembles de rayons.

Transformation affine : Soit $f = \lambda\vec{v}.A\vec{v} + B$ une transformation affine de $\mathbb{Q}^K$ dans lui même, où A est une matrice $n \times n$ et B est un n-vecteur de rationnels. Soit P un polyèdre, (S, R) son système générateur. Alors le polyèdre $f(P) = \{f(\vec{v})|\vec{v} \in P\}$ a pour ensemble de sommets l'ensemble $\{A\vec{s} + B|\vec{s} \in S\}$ et pour ensemble de rayons l'ensemble $\{A\vec{r}|\vec{r} \in R\}$.

Test du vide : Un polyèdre est vide si et seulement si il n'a pas de sommets.

Test d'inclusion et d'égalité : Un polyèdre P, de système générateur (S, R), est inclus dans un polyèdre Q, défini par le système de contraintes $A\vec{x} \leq B$ si et seulement si tout sommet $\vec{s} \in S$ satisfait $A\vec{s} \leq B$ et tout rayon $\vec{r} \in R$ satisfait $A\vec{r} \leq 0$.

La bibliothèque offre de plus :

- un opérateur d'élargissement, qui permet d'extrapoler la limite d'une suite croissante de polyèdres (cf. [CH78, Hal79, Hal93])
- un opérateur de “passage du temps”, utilisé pour les systèmes hybrides linéaires (voir §4.3), dans lesquels certaines variables évoluent continûment avec des dérivées soumises à des contraintes linéaires.

Inégalités strictes Les algorithmes de base n'autorisent que des polyèdres fermés, i.e., dont les contraintes sont des équations et des inégalités *larges*. Dans le cadre des automates hybrides, les variables manipulées varient continûment, et il est alors nécessaire de considérer des inégalités strictes. Nous avons étendu la bibliothèque [HPR94] pour prendre en compte de telles inégalités. Cette prise en compte ne demande que l'adjonction d'une nouvelle variable (ε) et une modification des tests du vide et de l'inclusion (où il est testé que ε est strictement positif).

4.2.3 Le calculateur de bureau $\boxed{\textbf{Tâche 2.7.3}}$

Le calculateur de bureau est un outil permettant d'invoquer de manière interactive les opérations de la bibliothèque. On peut ainsi créer des polyèdres identifiés comme des variables, et effectuer des opérations sur ces variables. Nous donnons simplement Fig. 3 un petit exemple de session, avec, en regard, les commentaires explicatifs.

4.3 L'analyseur d'automates hybrides linéaires $\boxed{\textbf{Tâche 2.7.3}}$

Nous avons appliqué l'interprétation abstraite basée sur les polyèdres [CH78, Hal79], à l'analyse et la vérification des "automates hybrides linéaires" [ACH+95, HPR94, HMP95]. Ces automates sont utilisés pour modéliser des systèmes combinant des aspects discrets et continus. Un automate hybride linéaire est un automate fini, combiné avec un ensemble de variables numériques évoluant continûment au cours du temps. A chaque état de l'automate sont associés un domaine des dérivées (un système de contraintes linéaires sur les dérivées des variables) et un invariant (un système de contraintes linéaires sur les variables). A chaque transition sont associés une garde (système de contraintes linéaires sur les variables) et un ensemble d'affectations linéaires (transformation affine). La sémantique intuitive est la suivante : lorsque l'automate est dans un état, les variables évoluent continûment au cours du temps, selon des dérivées constantes appartenant au domaine des dérivées associé à l'état, et ceci tant que leurs valeurs satisfont l'invariant associé. L'automate peut quitter un état par une transition discrète partant de cet état, à condition que les variables aient atteint des valeurs satisfaisant la garde de la transition. Les variables subissent alors les affectations discrètes associées à la transition, et le même comportement continue dans l'état but de la transition.

Sauf dans certains cas particuliers (cf. §4.4), l'analyse des automates hybrides linéaires pose des problèmes indécidables. C'est pourquoi une analyse approchée, à base d'interprétation abstraite, offre des perspectives particulièrement intéressantes dans ce domaine. D'autre part, notre analyse à base de polyèdres est spécialement adaptée à ce type de problème, pour les raisons suivantes :

- les contraintes linéaires apparaissent partout dans le modèle, et nous étions particulièrement bien armés pour les traiter.
- le comportement des variables continues est naturellement linéaire par morceaux, et notre analyse donne donc des résultats très précis dans ce contexte.
- la présence de variables continues force à traiter des ensembles denses, ce qui est justement le cas des polyèdres.

L'analyseur prend en entrée une description d'automate hybride linéaire, ou même une description structurée hiérarchique d'un tel automate en "hybrid Argos" [HMP95], sur laquelle il réalise une analyse approchée en avant : le résultat est, dans chaque état de l'automate, une approximation supérieure de l'ensemble des valeurs possibles des variables lorsque l'automate est dans l'état considéré. En particulier, lorsque cette approximation est le polyèdre vide, on peut en conclure que l'état en question est inaccessible. Ce type de résultat est utilisable en vérification.

```
polka 2 10
```

Invocation du calculateur en dimension 2 et avec 10 “variables polyèdres” au maximum.

```
Polka *** #1 := [x>=0 & y>=0 & x+y<=1];
[ x+y<=1 & y>=0 & x>=0 ];
[vertices (0 0) (0 1) (1 0)];
Polka *** #2 := [x>=0 & y>=0 & x<=1 &
y<=1];
[ y<=1 & x<=1 & y>=0 & x>=0 ];
[vertices (0 0) (0 1) (1 0) (1 1)];
Polka *** #3 := [x+y>=2 & 2x<=3 &
2y<=3];
[ 2y<=3 & 2x<=3 & x+y>=2 ];
[vertices (1 3)/2 (3 1)/2 (3 3)/2];
```

Création de 3 polyèdres, appelés #1, #2, et #3, à l'aide de leurs systèmes de contraintes. A chaque création, le calculateur fournit le système de contraintes éventuellement minimisé, et le système générateur correspondant.

```
Polka *** #4 := #1 hull #3;
[ x-y<=1 & y>=0 & 2x<=3 & 2y<=3 &
x-y>=-1 & x>=0 ];
[vertices (0 0) (1 3)/2 (3 1)/2 (3 3)/2
(0 1) (1 0)];
```

Calcul dans la variable #4 de l'enveloppe convexe des polyèdres #1 et #3.

```
Polka *** #5 := #2 inter #3;
[ x=1 & y=1 ];
[vertices (1 1)];
```

Calcul dans la variable #5 de l'intersection des polyèdres #2 et #3.

```
Polka *** #1 included #2;
Yes
```

Test d'inclusion de #1 dans #2.

```
Polka *** empty #1;
No
```

Test #1 est-il vide?

```
Polka *** quit
```

Arrêt du calculateur

Figure 3: Une session du calculateur de bureau

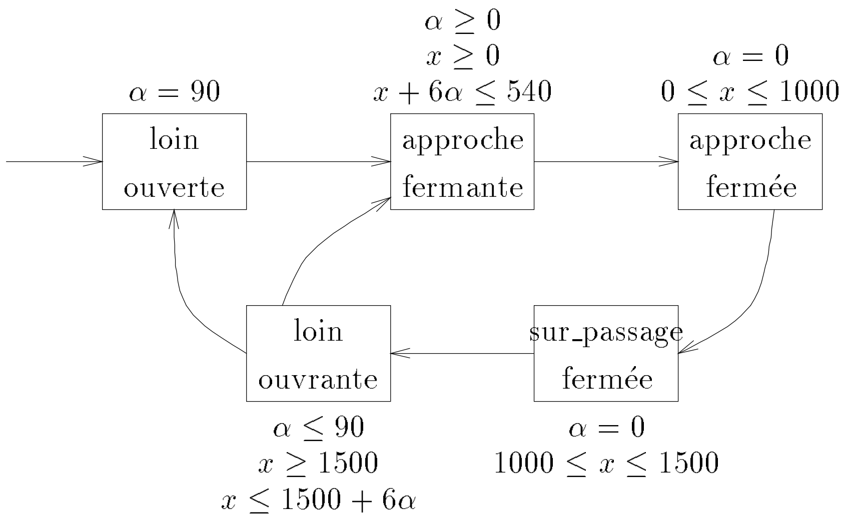


Figure 4: Résultats de l'analyse du "contrôle de passage à niveau"

A titre d'exemple, la Fig. 4 donne le résultat de l'analyse du célèbre "contrôle de passage à niveau": On considère un passage a niveau sur une voie de chemin de fer. 1000 mètres avant d'atteindre le passage, un train pousse une pédale qui prévient un contrôleur; celui-ci commande alors la fermeture de la barrière, laquelle se baisse à raison de 10° par seconde. 500 mètres après le passage, le train pousse une deuxième pédale, prévenant ainsi le contrôleur qu'il peut relever la barrière. Le problème est, sachant que la vitesse d'un train est inférieure à 60m/s, de montrer que la barrière est toujours fermée lorsqu'un train traverse le passage. La Fig. 4 montre uniquement la partie accessible de l'automate (et donc qu'aucun état "sur passage" où la barrière n'est pas "fermée" n'est accessible) avec les systèmes de contraintes trouvés : x est la distance du train à la pédale d'entrée, et α est l'angle de la barrière par rapport à l'horizontale.

4.4 Méthodes numériques : L'outil KRONOS

KRONOS [DOY94, DY95, DOTY96] est un outil d'aide à la conception de systèmes temporisés, c'est à dire soumis à des contraintes temporelles quantitatives. Des exemples de tels systèmes sont les protocoles de communication, les circuits et les systèmes multimédia. KRONOS vise à vérifier de manière automatique la correction du fonctionnement de ces systèmes.

Les systèmes temporisés sont décrits par des automates, dits automates temporisés [AD90, NSY92], augmentés d'un ensemble d'horloges qui mesurent l'écoulement du temps. Les comportements de l'automate sont restreints à ceux qui satisfont les contraintes sur les horloges associées aux transitions.

Les critères de correction dont la validation nous intéresse sont les propriétés temporelles quantitatives telles que par exemple l'absence de pannes dans un laps de temps borné et l'offre périodique d'un service particulier. Ces propriétés sont exprimées par des formules de la logique temporelle temps-réel TCTL [ACD90, HNSY92].

L'outil KRONOS fait l'objet d'une distribution publique par Internet. Il est utilisé dans des nombreux sites, par exemple en France, au CERT à Toulouse, au LAG à Grenoble et à l'INRIA Rhône-Alpes, et à l'étranger, notamment aux Etats Unis, à l'Université de Californie à Berkeley et au laboratoire de recherche de Intel.

Nous présentons dans cette section les développements réalisés récemment autour de l'outil KRONOS : une bibliothèque d'opérations de manipulation symbolique de contraintes temporelles, un calculateur de bureau permettant d'utiliser de manière interactive cette bibliothèque, un algorithme de réduction du nombre d'horloges et un algorithme de génération du modèle minimal.

4.4.1 Représentation symbolique par des matrices de bornes $\boxed{\textbf{Tâche 2.7.1}}$

Les algorithmes mis en œuvre dans l'outil KRONOS s'appuient sur une représentation symbolique des ensembles d'états par des contraintes linéaires particulières sur les horloges du système. Ces contraintes sont des comparaisons d'horloges, ou de différences d'horloges, à des entiers. Considérons par exemple la contrainte :

$$1 \leq x \leq 4 \wedge z \leq 5 \wedge z - x \leq 0 \wedge x - z \leq 2$$

Elle peut être représentée au moyen de la matrice suivante :

$$\left(\begin{array}{cccc} & \mathbf{0} & x & z \\ \mathbf{0} & (0,\leq) & (-1,\leq) & (0,\leq) \\ x & (4,\leq) & (0,\leq) & (2,\leq) \\ z & (5,\leq) & (0,\leq) & (0,\leq) \end{array}\right)$$

Chaque entrée de la matrice représente la borne supérieure de la différence entre deux horloges. Par exemple, l'entrée M_{xz} est égale à $(2,\leq)$ ce qui représente la contrainte $x-z \leq 2$. Il en va de même pour l'entrée M_{zx}.

Les entrées $M_{\mathbf{0}x}$ et $M_{x\mathbf{0}}$ représentent les bornes inférieures et supérieures de l'horloge x. $\mathbf{0}$ est une "horloge" fictive dont la valeur est toujours 0. Ceci permet de représenter la borne inférieure de x comme la différence $\mathbf{0} - x \leq -1$ et la borne supérieure de x comme la différence $x - \mathbf{0} \leq 4$. Il en va de même pour z. Comme les valeurs des horloges sont toujours positives, la borne inférieure de z est $(0,\leq)$. Les éléments de la diagonale sont toujours égaux à $(0,\leq)$.

La matrice M est appelée *matrice de bornes*. D'un point de vue géométrique, une matrice de bornes définit un polyèdre convexe dans l'espace des valuations des horloges qui est déterminé par les bornes supérieures et inférieures pour les valeurs des horloges, et des bornes supérieures pour les valeurs des différences entre deux horloges (cf. Fig. 5).

4.4.2 Les matrices de bornes

L'ensemble des bornes, noté $\mathcal{B}$, est défini par :

$$\mathcal{B} = \mathbb{Z} \times \{<,\leq\} \bigcup \{(-\infty,<),(\infty,<)\}$$

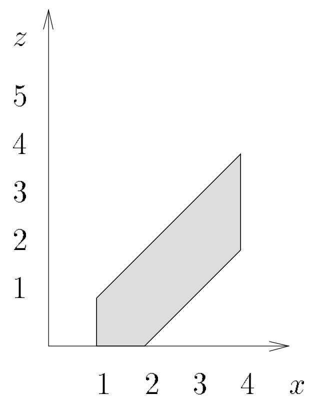


Figure 5: Polyèdre correspondant à une matrice de bornes

Les symboles de relation $<$ et $\leq$ sont totalement ordonnés : $<$ est supposé strictement plus petit que $\leq$. L'ordre total entre les symboles de relation et l'ordre total $\leq$ sur les entiers augmentés de $\{-\infty, \infty\}$ de façon naturelle, induit un ordre total $\sqsubseteq$ sur l'ensemble $\mathcal{B}$ des bornes. Pour tout $(c_1, \prec_1), (c_2, \prec_2) \in \mathcal{B}$, on définit : $(c_1, \prec_1) \sqsubseteq (c_2, \prec_2)$ si $c_1 < c_2$, ou si $c_1 = c_2$, si $\prec_1$ est inférieur ou égal à $\prec_2$. Notons que la borne $(\infty, <)$ satisfait, pour tout $b \in \mathcal{B}$, $b \sqsubseteq (\infty, <)$. Pour toute borne $(c, \prec) \in \mathcal{B}$, $\neg(c, <)$ est la borne $(-c, \leq)$ et $\neg(c, \leq)$ est la borne $(-c, <)$.

Pour tout $(c_1, \prec_1), (c_2, \prec_2) \in \mathcal{B}$, on définit :

$$(c_1, \prec_1) \otimes (c_2, \prec_2) = (c_1 + c_2, \min(\prec_1, \prec_2))$$

et

$$(c_1, \prec_1) \oplus (c_2, \prec_2) = \min((c_1, \prec_1), (c_2, \prec_2))$$

Soit $\mathcal{H}$ l'ensemble d'horloges et $\mathcal{H}^\sharp = \mathcal{H} \bigcup \{\mathbf{0}\}$. Soit M une *matrice carrée de bornes* sur $\mathcal{H}^\sharp$. L'*ensemble caractéristique* de M, noté $[\![M]\!]$, est l'ensemble des valuations $v \in [\mathcal{H}^\sharp \to \mathbb{R}^+]$ telles que pour tout $x, y \in \mathcal{H}^\sharp$, $v(x) - v(y) \prec c$ où $M_{xy} = (c, \prec)$. Afin de simplifier la notation, par la suite on écrit $v(x) - v(y) \prec M_{xy}$. L'ordre total $\sqsubseteq$ défini sur les bornes peut être étendu de manière naturelle aux matrices de bornes. Pour toute paire de matrices M et M', $M \sqsubseteq M'$ si pour tout $x, y \in \mathcal{H}^\sharp$, $M_{xy} \sqsubseteq M'_{xy}$. Par conséquent, si $M \sqsubseteq M'$ alors $[\![M]\!] \subseteq [\![M']\!]$.

Les opérations $\otimes$ et $\oplus$ peuvent également être étendues aux matrices de bornes de la façon suivante :

$$(M \otimes M')_{xy} = \bigoplus_{z \in \mathcal{H}^\sharp} M_{xz} \otimes M'_{zy} \text{ et } (M \oplus M')_{xy} = M_{xy} \oplus M'_{xy}$$

Posons $\mathbf{e} = (0, \leq)$ et $\mathbf{n} = (\infty, <)$. On définit les matrices $\mathbf{N}$ et $\mathbf{E}$: $\mathbf{E}_{xy} = \mathbf{e}$ si $x = y$, $\mathbf{E}_{xy} = \mathbf{n}$ sinon, et $\mathbf{N}_{xy} = \mathbf{n}$. Notons que $\mathbf{E}$ est l'élément *neutre* pour $\otimes$ et $\mathbf{N}$ est l'élément *absorbant* pour $\otimes$ et l'élément *neutre* pour $\oplus$.

Soit $\mathcal{M}$ l'ensemble des matrices de bornes défini par :

$$\mathcal{M} = \{M \mid \forall x \in \mathcal{H}^\sharp.\ M_{xx} \sqsubseteq (0, \leq)\} \bigcup \{\mathbf{N}\}$$

On définit : $M^* = \bigoplus_{k \in \mathbb{N}} M^k$ où $M^0 = \mathbf{E}$ et $M^{k+1} = M^k \otimes M$. La structure $(\mathcal{M}, \oplus, \otimes, \mathbf{N}, \mathbf{E})$ est une *algèbre régulière*.

Opérations symboliques.

Ensemble vide. L'ensemble vide est représenté de manière unique par la matrice $\mathbf{O}$ telle que pour tout $x, y \in \mathcal{H}^\sharp$, $\mathbf{O}_{xy} = (-\infty, <)$.

Forme canonique. Plusieurs matrices différentes peuvent représenter la même contrainte. Toutefois, il en existe une canonique. Pour toute matrice $M \in \mathcal{M}$, la forme canonique de M est la matrice définie par : $\mathbf{cf}(M) = \mathbf{O}$ si $[\![M]\!] = \emptyset$, M^* sinon. La matrice $\mathbf{cf}(M)$ peut être calculée par l'algorithme de Floyd-Warshall. Ceci permet aussi de déterminer si l'ensemble caractéristique de M est vide. En effet, si la matrice obtenue par application de l'algorithme de Floyd-Warshall est telle qu'un élément de la diagonale est inférieur à $(0, \leq)$, l'ensemble caractéristique de M est vide.

Conjonction. Si ψ et ψ' sont deux contraintes représentées par les matrices M et M', alors la contrainte $\psi \wedge \psi'$ est représentée par la matrice $M \oplus M'$. Autrement dit, l'opération $\oplus$ sur les matrices de bornes coïncide avec l'intersection des ensembles caractéristiques.

Matrice universelle. L'ensemble de toutes les valuations est définit par la matrice $\mathbf{U} \subseteq \mathcal{M}$ telle que pour tout $x, y \in \mathcal{H}^\sharp$, $\mathbf{U}_{xy} = (0, \leq)$ si $x = y$ ou $x = \mathbf{0}$, $\mathbf{U}_{xy} = (\infty, <)$ sinon.

Test du vide. La contrainte ψ représentée par la matrice M est fausse si et seulement si $\mathbf{cf}(M) = \mathbf{O}$.

Autres opérations. D'autres opérations nécessaires à la vérification ont également été implantées dans la bibliothèque. Il s'agit par exemple des opérateurs de remise à zéro des horloges et de passage du temps [HNSY92].

4.4.3 Le calculateur de bureau

Tâche 2.7.3

Nous avons mis en œuvre un calculateur de bureau permettant d'utiliser de manière interactive toutes les opérations définies sur les matrices de bornes. La table 1 présente un exemple de session de cet outil.

`?  #clocks = 2`	Définition d'un ensemble d'horloges de dimension égale à deux.
`?  #zones = 10`	Création de dix variables de type matrice.
`clocks = x1 x2`	Le calculateur annonce que les horloges sont
`zones = z0 z1 z2 z3 z4 z5 z6 z7 z8 z9`	x1 et x2, et les variables z0 à z9.
`?  z0 := x1>=1 & x1<=4 & x2<=x1 &`	On introduit la contrainte de l'exemple.
`x1<=x2+2 & x2<=5.`	
`?  cf z0.`	On calcule la forme canonique de z0.
`?  print z0.`	On imprime la matrice z0.
`(0,1) (-1,1) (0,1)`	Le second élément de la paire représente le
`(4,1) ( 0,1) (2,1)`	symbole de relation : 0 pour $<$, 1 pour $\leq$.
`(4,1) ( 0,1) (0,1)`	
`?  show z0.`	Imprimer la contrainte représentée par z0.
`1<=x1 and x1<=4 and x1<=x2+2 and x2<=x1`	
`?  z1 := z0 & x1<1.`	Calculer l'intersection de z0 avec $x1 < 1$.
`?  cf z1.`	Calculer la forme canonique de z1.
`?  show z1.`	Imprimer z1.
`FALSE`	La contrainte est fausse.
`?  end.`	Fin de la session.

Table 1: Calculateur de bureau des matrices de bornes.

4.4.4 Réduction du nombre d'horloges $\boxed{\text{Tâche 2.7.4}}$

Nous avons développé et mis en œuvre un algorithme pour réduire le nombre d'horloges d'un automate temporisé [DY96]. En effet, nous avons constaté que, dans la pratique, un système contient souvent des nombreuses horloges redondantes. Etant donné que la complexité du problème de la vérification de systèmes temporisés est exponentielle dans le nombre d'horloges, il est nécessaire de les éliminer.
Deux sources de redondance sont :

- la génération automatique d'automates temporisés par les compilateurs de langages de haut niveau
- la mise en parallèle

En partant de ces constatations, nous avons développé un algorithme qui combine deux stratégies complémentaires. La première consiste à détecter des horloges dont les valeurs ne sont pas testées par les contraintes associées aux transitions de l'automate. La deuxième consiste à détecter des paires d'horloges qui sont toujours égales.

Les résultats expérimentaux obtenus avec des études de cas significatives montrent que la réduction achevée par l'algorithme conduit à des économies d'espace mémoire de presque 50% en moyenne, pouvant aller jusqu'à plus de 90% dans certains cas.

4.4.5 Génération d'un modèle minimal $\boxed{\text{Tâche 2.7.4}}$

Nous avons développé et implanté un algorithme qui, étant donné un automate temporisé, génère son modèle quotient modulo une relation d'équivalence entre les états qui fait abstraction du temps exact écoulé [TY96]. Cette relation est une bisimulation qui préserve une grande partie des propriétés temporelles que l'on veut vérifier, en particulier celles de sûreté.

D'une part, les résultats expérimentaux obtenus avec des études de cas significatives montrent que cet algorithme est plus performant que d'autres proposés précédemment, notamment à l'Université de Stanford. D'autre part, notre travail a créé un lien entre KRONOS et la boîte à outils CADP développée à l'IMAG. En effet, étant donné que le modèle quotient généré peut être vu comme un système de transitions étiquetées non-temporisé, il est désormais possible d'utiliser les commandes BCG pour le visualiser, ainsi que l'outil ALDEBARAN pour vérifier des propriétés de sûreté.

4.5 Construction de programmes abstraits pour la vérification de systèmes distribués $\boxed{\text{Tâche 2.7.4}}$

Notre but est de vérifier des propriétés exprimables par des formules d'une logique temporelle (ici nous considérons le μ-calcul arborescent [Koz83]) à l'aide de techniques de model-checking sur le *modèle* (graphe d'exécution) du programme à vérifier. Le principal problème de ce type de méthodes est qu'en général ce modèle est trop grand pour être représenté de manière explicite. Les méthodes de représentation implicite sous formes de BDD ou polyèdres permettent de repousser les limites de cette méthodes, mais elles ne s'appliquent pas à des programmes manipulat des types de données comme les files de messages, qui sont utilisées couramment dans les systèmes distribués. Une solution consiste à vérifier, à la place du programme initial, un programme abstrait, plus simple, et dont toutes les variables sont de type fini. La question qui se pose est “quelles sont les propriétés dont la satisfaction sur le programme abstrait implique la satisfaction sur le programme concret ?”.

4.5.1 Préservation de propriétés

Les programmes que nous considérons représentent des systèmes distribués qui partagent des variables (les canaux de transmission de messages). Un système S est un ensemble de processus P_i qui s'exécutent en parallèle. Chaque processus est un triplet $(V, T, init)$ où :

- $V = (v_1, \ldots, v_k)$ est un vecteur de variables d'états (locales ou partagées)
- $T = \{\tau_1, \ldots, \tau_p\}$ est un ensemble de transitions sous forme de commandes gardées, c'est-à-dire, chaque transition τ est de la forme

  $$g_\tau(V) \longrightarrow R_\tau(V, V')$$

  où le prédicat g_τ représente la garde, c'est-à-dire, la condition sous laquelle la transition est exécutable, et le prédicat R_τ représente la relation de transition.
- $init$ est le prédicat représentant l'ensemble des états initiaux possibles.

Si les gardes de plusieurs transitions sont satisfaites dans un état, le choix de la transition effectivement chosie (et l'état successeur atteint) se fait de manière non-déterministe. Les programmes concrets qu'on considère sont en général déterministes, au moins dans le sens que toutes les relations R_τ sont des fonctions ; par contre pour les programmes abstraits, ceci n'est en général pas le cas.

Une abstraction d'un processus P_i est un processus ${P^A}_i$ où :

- $V^A = (v_1^A, \ldots, v_l^A)$ est un vecteur de variables d'états de types finis relié au vecteur de variables concret par un prédicat

  $$\rho(V, V^A)$$

  exprimant quels sont les états concrets représentés par chaque état abstrait. En pratique ρ représente souvent une fonction des états concrets dans les états abstraits.
- $T^A = \{\tau_1^A, \ldots, \tau_p^A\}$ est un ensemble de transitions abstraites de la forme

  $$g_\tau^A(V^A) \longrightarrow R_\tau^A(V^A, V^{A\prime})$$

  tel que pour toute transition τ

  $$(\exists V, V' \,.\, \rho(V, V^A) \wedge \rho(V', V^{A\prime}) \wedge R_\tau(V, V')) \quad \Rightarrow \quad R_\tau^A(V^A, V^{A\prime}) \quad (*)$$
- $init^A$ satisfait la condition

$$(\exists V \,.\, \rho(V, V^A) \wedge init(V)) \quad \Rightarrow \quad init^A(V^A)$$

c'est-à-dire, le programme abstrait représente sur le domaine du processus concret une relation de transition plus grande que le processus concret.

En utilisant les résultats de [CC77a], nous avons montré des résultats de préservation de propriétés assez généraux, disant que "toute formule f du μ-calcul sans négations (ou ayant des négations seulement sur les propositions atomiques) exprimant seulement des quantifications universelles sur les chemins sont préservés du programme abstrait vers le programme concret".

En outre, nous avons trouvé des conditions sous lesquelles un système, obtenu en définissant un processus abstrait pour chaque processus concret, représente une abstraction du système concret. S'il n'y a pas de variables partagées, c'est toujours le cas. Autrement, cela dépend de l'opérateur de mise en parallèle utilisé. Pour l'opérateur de mise en parallèle par entrelacement (une transition du système global est une transition d'un de ses processus), on doit juste exiger que l'abstraction des variables partagées est indépendante des valeurs de variables locales, et la même pour tous les processus qui partagent ces variables. Dans le cas de l'opérateur de mise en parallèle "synchrone" (où chaque transition du système global est composée d'une transition simultanée de chaque processus) on doit de plus exiger que l'abstraction des variables partagées représente une fonction du domaine des variables partagées du système concret vers les variables partagées du système abstrait. Ces résultats sont publiés dans [LGS+95].

4.5.2 Abstraction de programmes booléens

Dans le cas où le domaine des variables du programme concret est fini, le processus de calcul de programme abstrait peut être automatisé complètement, et en plus on peut construire le programme abstrait le plus précis pour une relation d'abstraction donnée en éliminant les quantificateurs du côté gauche de l'implication (*). Dans sa thèse [Loi94], Claire Loiseaux a implémenté un outil qui, à partir d'un vecteur de variables booléennes abstrait et d'une relation d'abstraction ρ, construit, pour chaque processus concret P_i sur des variables booléennes, le processus abstrait le plus précis satisfaisant la condition (*), en utilisant des BDD pour le codage de chaque transition concrète et abstraite, ainsi que de la relation ρ.

Puisque, en général, chaque transition dépend seulement d'un petit nombre de variables, et que la relation ρ est telle que la valeur de chaque variable abstraite dépend seulement de peu de variables concrètes (souvent une seule), nous avons pu calculer et analyser des systèmes abstraits pour des systèmes de taille assez importante, comme par exemple un protocole de dépassement de véhicules décrit en [EFJ90] (le programme concret contient presque 200 variables booléennes).

4.5.3 Abstractions de programmes à domaine infini

Quand le domaine du système concret n'est pas fini, on ne peut en général pas construire un système abstrait comme ci-dessus. Mais considérons le cas où la relation ρ est telle que la valeur de chaque variable abstraite dépend de la valeur d'une seule variable concrète, et que le nombre de "prédicats atomiques" (dans le cas de variables entières par exemple *is_zero, is_prime, greater_than,...*) et d'opérations atomiques (comme $+$, $*$,...) est relativement restreint. Dans ce cas, on peut essayer d'associer à chaque occurrence d'un prédicat (ou opération) p atomique un prédicat (ou une opération) p^A sur le domaine abstrait correspondant, et montrer que la condition (*) est vraie pour chaque paire (p, p^A) (c'est-à-dire p^A est une interprétation abstraite de p). Dans le cas, où la définition du système concret ne contient pas de négations, le système abstrait obtenu en remplaçant chaque occurrence d'un prédicat (ou d'une opération) atomique par son abstraction, et chaque variable par la variable abstraite correspondante, est une abstraction du système concret.

En procédant de cette sorte, nous avons montré [Gra95] la correction de la mémoire cache, réputée difficile à vérifier, décrite dans [ABM93]. Nous avons utilisé la boîte à outils CAESAR/ALDÉBARAN [FGK+96]. La boîte à outils permet de vérifier des systèmes décrits en LOTOS où les types des données et leurs prédicats et opérations associés sont décrits par des types abstraits, ce qui a particulièrement facilité la définition du programme abstrait.

5 Travaux effectués au LIENS

Le groupe "Sémantique et interprétation abstraite" du LIENS travaille sur la sémantique et l'analyse sémantique de programmes par interprétation abstraite [Cou96].

La sémantique consiste à définir les comportements possibles d'un programme à l'exécution pour un langage de programmation donné. Les efforts dans ce domaine ont porté à la fois sur les méthodes formelles de description de sémantique, principalement en étendant l'extension des descriptions de sémantiques opérationelles aux comportements infinis ([CC92c, CC93d] basée sur [CC92d]) et sur la description du parallélisme (dite *vraiment parallèle* par opposition à une simulation non-déterministe sur un monoprocesseur) grâce à des automates de dimensions supérieures [Gou93a, Gou95a, Gou96a, Gou96b].

Le travail sur l'analyse de programmes comporte une partie fondamentale concernant les bases théoriques et des applications à différentes sortes de languages et de propriétés de l'exécution.

En ce qui concerne les bases théoriques de l'interprétation abstraite, le travail a porté sur la présentation compositionnelle [CC95a], sur la notion de complétude [Cou95c], sur la comparaison des analyses basées sur des correspondances de Galois et des opérateurs d'élargissement [CC92b, CC93c] et sur le développement du modèle des correspondances de Galois à l'ordre supérieur [CC94].

Les travaux sur les applications de l'interprétation abstraite ont été nombreux. Ils concernent l'analyse de nécessité (*strictness analysis*) pour les languages fonctionnels. Nous avons montré que cette analyse se formule très simplement en utilisant des correspondances de Galois [CC93b], sans avoir besoin de recourir à des modèles plus faibles tout en permettant de regrouper diverses analyses du même type [CC94]. Ceci s'applique également à l'analyse des temps de liaison (*binding-time analysis*), ce qui permet de combiner les deux analyses [Ved95]. Enfin, il est possible d'intégrer l'analyse de nécessité dans un système de types [Mon95b]. Bien que très populaire, les nombreuses implantations naïves de l'analyse de nécessité sont beaucoup trop coûteuses pour les languages d'ordre supérieur pour être d'une quelconque utilité pratique. En utilisant un module de graphes de décision binaires (BDD), il est possible d'en faire une implantation efficace [Mau94].

Une seconde application importante du travail sur l'analyse de programmes a concerné l'analyse de programmes parallèles [CC93a, Cri95, CG93, Gou95b] et la combinaison des méthodes d'interprétation abstraite et de vérification des systèmes finis [Cou95a].

Enfin, si le travail sur les domaines abstraits a principalement porté dans le passé sur les domaines numériques [Mas93a, Mas93b], les applications récentes concernent des propriétés symboliques comme les grammaires utilisées dans les analyses dites ensemblistes (*set-based analysis*) [Cou95e, CC95b] ou le domaine des termes de Herbrand utilisé pour l'analyse de programmes logiques [Cou95b] et le typage [Cou97, Mon92a, Mon92b, Mon93a, Mon93b, Mon94, Mon95a, Mon95b].

Ces travaux sont décrits dans les sous-sections qui suivent :

5.1 Analyse de comportement $\boxed{\text{Tâche 2.7.1}}$

Le terme analyse de comportement regroupe diverses analyses pour les languages fonctionnels, comme l'analyse de nécessité, l'analyse des temps de liaison, l'analyse de terminaison, les analyses à base de projections (*projection analysis*) et leurs duales, les analyses à base de relations d'équivalence partielle (PER-*analysis*, l'analyse des temps de liaison (*binding-time analysis*), etc. [CC94].

5.2 Utilisation des graphes de décision binaires en interprétation abstraite $\boxed{\text{Tâche 2.7.4}}$

L'idée principale de ce travail [Mau94] est d'utiliser une représentation symbolique des fonctions booléennes pour construire des interprétations abstraites plus performantes. Les représentations utilisées sont les TDG [Bil87a], une variante des BDD. Ce sont des graphes de décision, c'est-à-dire que pour avoir les valeurs d'une fonction, on parcourt le graphe en choisissant son chemin selon les valeurs des variables. Chaque nœud représente une variable, ou un test sur cette variable. Un TDG représente une fonction booléenne associée à une liste de variables ordonnées. Dans la plupart des cas, cette représentation est très compacte. De plus, la plupart des opérations sur les TDG peuvent être très rapides, en particulier pour les opérateurs orthogonaux qui suivent la structure des TDG. Une autre particularité intéressante est qu'on peut utiliser les TDG eux-mêmes pour représenter les opérations sur les TDG.

En interprétation abstraite, les sémantiques sont prises dans des domaines de sémantique. Les domaines de la sémantique abstraite sont souvent représentables par ordinateur, et donc codables sous forme de TDG. On peut vouloir coder, selon l'efficacité des codages trouvés le domaine abstrait, les fonctions abstraites[3] ou mieux encore, les deux. Dans le cas de deux domaines abstraits codés, on peut coder en toute généralité le domaine abstrait des fonctions de l'un dans l'autre, pourvu qu'on puisse associer à chaque variable du premier domaine une arité maximum des TDG qu'elle représente. Cela fournit donc aussi un codage des fonctions abstraites.

Il existe des fonctions pour lesquelles la représentation sous forme de TDG n'est pas compacte, ce qui peut consommer énormément d'espace mémoire, mais aussi de temps,

[3]Elles servent à calculer la sémantique particulière d'un programme.

car la plupart des opérations sur les TDG dépendent de leur taille. La solution spécifique à l'interprétation abstraite est d'approximer ces fonctions par d'autres prenant moins de place. Cette approximation peut servir dans le cadre d'un opérateur d'élargissement dynamique si le domaine abstrait est codé, ou encore elle peut servir à approximer dès le départ les fonctions abstraites elles-mêmes, accélérant chaque itération. Cette approximation peut être faite en temps polynomial, mais reste très lente.

Comme exemple d'utilisation de ces différents procédés, on a codé l'analyse de nécessité, pour aboutir à un algorithme très performant. En effet, dans ce cas précis, l'analyse de Mycroft donne déjà le domaine abstrait sous forme d'ensemble de fonctions booléennes, et l'ordre d'approximation est très simple à calculer.

5.3 Combinaison d'analyses de comportement $\boxed{\text{Tâche 2.7.4}}$

Une analyse de comportement, basées sur [Ved95], combinant une analyse de terminaison, une analyse de necessité, une analyse des temps de liaisons et une analyse de typage d'ordre supérieur pour un langage fonctionnel simple est en cours d'implantation. L'implémentation est à l'heure actuelle en Java, et sera éventuellement transcrite en C++ si jamais cela s'avère trop lent.

5.4 Analyse de programmes parallèles $\boxed{\text{Tâche 2.7.1}}$

Sémantique vraiment parallèle De nombreux modèles sémantiques sont basés sur des systèmes de transition non déterministes. Ceci revient à faire une première approximation qui consiste à supposer que le programme parallèle s'exécute sur un seul processeur multiprogrammé. Cette hypothèse est sans conséquence pour certaines propriétés comme l'invariance mais fatale pour d'autres comme le degré maximal de parallélisme. De plus, le mélange non-déterministe revient à faire un développement combinatoire des exécutions possibles très coûteux pour l'analyse: dire *a parallèle b* est quand même plus concis que *a suivi de b ou b suivi de a*.

Pour servir de fondement aux analyses parallèles, il était nécessaire de commencer par un travail sur des sémantiques vraiment parallèles qui permettent de définir de façon dénotationnelle ou bien opérationnelle des langages parallèles aussi divers que Linda, Concurrent Pascal ou Concurrent ML [Gou93a, Gou95a]. Ensuite, par interprétation abstraite, on peut construire effectivement un automate approché qui modélise les comportements dynamiques possibles du programme considéré, et permet d'analyser par vérification de modèles finis abstraits (*abstract model-checking*) [Cri96a, Cri96b, CG93].

L'intérêt de l'équipe a porté sur des sémantiques qui décrivent véritablement le parallélisme entre processus concurrents par des automates munis de transitions de dimension supérieure. Cette dimension décrit le degré de parallélisme effectif lors de l'exécution d'opérations concurrentes. Cette information supplémentaire est importante pour effectuer certaines analyses de programmes comme l'exclusion mutuelle ou l'indépendance et valider des propriétés liées à la géométrie des systèmes de transitions (blocages, branchements, etc.).

On peut définir diverses sortes de systèmes de transitions de dimension supérieure. Les systèmes totaux (HTS) décrivent toutes les transitions de dimension supérieure qui vérifient une propriété de confluence locale. Ils permettent de modéliser la confluence et le branchement à tout niveau de parallélisme. Les systèmes partiels (PHTS) permettent de plus de modéliser l'interférence entre plusieurs actions parallèles.

Interprétation abstraite de programmes parallèles Ces automates permettent de définir des sémantiques approchées, calculables par interprétation abstraite en vue de l'analyse statique des programmes parallèles considérés. Cela a été réalisé pour différents types de langages (Linda [CG93], Concurrent Pascal [Cri96a], CML [Cri96b]) et une propriété de correction sémantique de l'interprétation abstraite a été prouvée. Deux techniques différentes permettent d'atteindre cet objectif :

- L'approche dénotationnelle vise à calculer l'automate abstrait de manière compositionnelle, en donnant des équations sémantiques structurées sur la syntaxe. Cela fonctionne pour des algèbres de processus comme CCS ou le Linda-calcul qui a été défini dans l'équipe. Mais les équations se compliquent lorsqu'un état partagé entre les divers processus est à prendre en compte.

- C'est pourquoi une approche opérationnelle a été préférée pour un langage à mémoire partagée comme Concurrent Pascal. Il suffit alors de donner les règles définissant les transitions à grain fin autorisées entre deux configurations. Les transitions de dimension supérieure se construisent aisément à l'aide des transitions de dimension plus basse.

L'implantation d'un analyseur de Concurrent Pascal basé sur une telle sémantique a été effectuée. Certaines techniques d'implémentation comme l'utilisation de transitions d'un grain plus grossier que dans les définitions sémantiques (Virtual Coarsening) permettent l'interprétation abstraite précise de programmes de taille raisonnable car elles diminuent fortement (au moins d'un facteur 4) le nombre d'états à considérer. Pour des algorithmes classiques d'exclusion mutuelle comme celui de Dekker, la preuve d'invariance est ainsi réalisée par l'interprète abstrait en quelques secondes de calcul. L'automate abstrait qui en résulte compte un millier d'états alors que le nombre d'états réels de ce programme est de l'ordre de cent mille. On gagne donc un facteur 100 sur la vérification de la propriété d'exclusion mutuelle dans cet exemple significatif.

Analyse des ordonnanceurs de programmes parallèles et de machines distribuées par interprétation abstraite Les ordonnanceurs (schedulers) de programmes parallèles ou distribués sont le sujet de [Gou95b]. Dans cet article, il est montré comment partir d'une sémantique d'un programme parallèle ou d'une machine distribuée en termes d'automates de dimension supérieure (une extension vraiment parallèle des systèmes de transitions ordinaires [Gou93a, Gou95a, Gou96a, Gou96b]), et calculer à partir de cette description, par une abstraction de cette sémantique, l'ensemble des ordonnanceurs "essentiels" de ce programme. Ces ordonnanceurs sont tout simplement les ordres possibles d'exécution des actions du programme modulo les commutations d'actions. Ces commutations sont monnaie courante dans les machines parallèles, elles correspondent au fait que deux actions n'interfèrent pas et donc que leurs ordres respectifs d'exécution sont indifférents quand on veut déterminer ce que calcule effectivement le programme. Ces ordonnanceurs "essentiels" déterminent donc précisément les comportements possibles d'un programme parallèle ou distribué. En particulier ils permettent de vérifier (partiellement) ces programmes. Un cas typique est celui des protocoles gérant des réseaux complexes de machines. Prouver par calcul abstrait (sur la sémantique abstraite des ordonnanceurs) qu'un protocole a tous ces ordonnanceurs appartenant à un certain ensemble de comportements séquentiels prédeterminés revient à prouver qu'il se comporte selon une certaine spécification séquentielle.

Ces résultats ont été également généralisés à certains comportements sur un nombre fini de processeurs (supérieur ou égal à un, cas précédemment développé). Cela permet de spécifier plus finement les comportements des protocoles, en présence (ou en l'absence) de processeurs.

On a commencé une implémentation de l'algorithme de vérification de protocoles décrit dans [Gou95b] en langage C. L'idée est de représenter la sémantique abstraite par des grandes matrices creuses décrivant les différentes transitions. Le calcul de relations de dépendances entre les lignes de ces matrices fournit des représentants de traces (ou ordonnanceurs) modulo les commutations d'actions autorisées. Pour calculer ces relations de dépendances, on se sert d'algorithmes développés par les cryptographes pour des problèmes de cribles quadratiques.

```
(* valabs.mli *)
(* valeurs abstraites *)
(* Regis Cridlig *)
(* 20/4/96 *)

exception Erreur_execution of string

val debug: bool ref

(* grace au typage statique, somme des domaines type's *)

type borne = Int of int | Infmoins | Infplus

val min: borne * borne -> borne  (* sur 2 bornes gauches *)
val max: borne * borne -> borne  (* sur 2 bornes droites *)

type valeur =
    Ent of borne * borne  (* borne inf et borne sup *)
  | Bool of bool
  | Booltop  (* true U false *)
  | Tableau of int * valeur array (* les bornes sont fixes pour le type *)

val entier: int -> valeur              (* repre'sentation d'un entier *)
val entiertop: valeur                  (* le top des entiers *)

val plus: valeur * valeur -> valeur     (* de'fini sur 2 entiers *)
val moins: valeur * valeur -> valeur    (* de'fini sur 2 entiers *)
val mult: valeur * valeur -> valeur     (* de'fini sur 2 entiers *)
val quotient: valeur * valeur -> valeur
val egal: valeur * valeur -> valeur     (* de'fini sur 2 entiers *)
val lt: valeur * valeur -> valeur       (* de'fini sur 2 entiers *)
val oppose: valeur -> valeur            (* oppose' d'un entier abstrait *)
val non: valeur -> valeur               (* ne'gation d'un boole'en abstrait *)
val et: valeur * valeur -> valeur       (* de'fini sur 2 boole'ens abstraits *)
val ou: valeur * valeur -> valeur       (* de'fini sur 2 boole'ens abstraits *)
val zerop: valeur -> valeur

val tableau_val: valeur -> int * valeur array
val out_valeur: out_channel -> valeur -> unit
val affiche_valeur: valeur -> unit
val union_valeur: valeur * valeur -> valeur (* borne sup de 2 valeurs *)
```

Figure 6: Signature du module du calcul d'intervalles d'entiers

5.5 Modules d'implantation de domaines abstraits $\boxed{\text{Tâche 2.7.1}}$

```
(* karr.mli *)
(* implemente le treillis des relations affines de Karr *)
(* Re'gis Cridlig *)
(* 9/6/96 *)

open Array
open Num
open Variable (* definit un type "variable" *)

type karr = num array array (* tableau de lignes *)
* variable array (* vecteur des variables-colonnes *)

val canonize : karr -> karr

val glb : karr -> karr -> karr
val lub : karr -> karr -> karr

val print_karr : karr -> unit
```

```
(* karrlist.mli *)
(* implemente le treillis des relations affines de Karr *)
(* a' l'aide de listes *)
(* Re'gis Cridlig *)
(* 20/8/96 *)

module type VariableType =
sig
type variable

val compare : variable * variable -> int
val string_of_var : variable -> string
val special : variable (* variable speciale superieure aux autres
*)
end

module Make(Var: VariableType) :
sig
open Num

open Var (* definit un type "variable" avec un test "compare",
une fonction string_of_var, et special ¿ autres variables *)

type karr = (variable * num) list list
(* tableau de lignes x colonnes *)
(* les variables sont trie'es *)
(* on represente un espace affine de demiension n
par un espace lineaire de dimension n+1 (coord. speciale) *)

val canonize : karr -> karr
val glb : karr -> karr -> karr
val lub : karr -> karr -> karr
val print_karr : karr -> unit

(* peut lever Not_found *)
val first_non_zero : (variable * num) list -> variable * num
val set_nth : 'a list -> int -> 'a -> 'a list
val del_nth : 'a list -> int -> 'a list
val top : karr
end
```

Figure 7: Signature des modules des relations affines entre entiers

Différents modules implantant des domaines abstraits ont été écrits en ML par Régis Cridlig pour être utilisés dans les analyseurs sémantiques. Ils incluent un module de calcul sur des intervalles d'entiers $[a, b]$, $[a, +\infty[$, $]-\infty, b]$ ou $]-\infty, +\infty[$ (où a et b sont des valeurs entières); deux modules de calcul sur des ensembles de relations affines entre variables entières [Ka76]; et un module de calcul sur les automates de dimension supérieure décrits plus hauts.

5.6 Interfaces utilisateurs $\boxed{\text{Tâche 2.7.2}}$

Pour utiliser les résultats d'une analyse, il faut disposer d'une interface utilisateurs souple et interactive. En particulier, ce besoin se fait sentir pour la mise au point abstraite, qui est en tous points semblable aux techniques de test ou de vérification de modèles finis (*model-checking*) classiques, sauf que l'on utilise des valeurs abstraites au lieu de valeurs concrètes, ce qui revient à effectuer simultanément un ensemble infini de jeux tests, au prix d'une perte d'information. Les interfaces classiques développées avec des interfaces graphiques non portables et de bas niveau comme *X11* [Han96] ou *Windows* semblent désormais n'être qu'une solution de fortune l'apparition récente d'environnements distribuées portables comme *HTML* ou *Java*. La programmation distribuée permet de séparer l'interface de l'analyseur statique proprement dit, ce qui permet d'envisager l'intégration aisée du même outil d'analyse sémantique dans un compilateur, un metteur au point abstrait, un outil de vérification, etc. Ce point de vue est développé dans [Cri96a].

```
(* relation.mli *)
(* relations over ordered types *)
(* all modifications are purely applicative *)
(* Regis Cridlig 1994-1996 *)
(* last modified 19/3/96 *)

module type OrderedType =
  sig
    type t
    val compare: t -> t -> int
  end
          (* The input signature of the functor [Relation.Make].
             [t] is the type of the keys.
             [compare] is a total ordering function over the keys.
             This is a two-argument function [f] such that
             [f e1 e2] is zero if the keys [e1] and [e2] are equal,
             [f e1 e2] is strictly negative if [e1] is smaller than [e2],
             and [f e1 e2] is strictly positive if [e1] is greater than [e2].
             Examples: a suitable ordering function for type [int]
             is [(-)]. You can also use the generic structural comparison
             function [compare]. *)

module type S = sig
  type key
          (* The type of the keys. *)

  type 'b r
        (* The type of relations from type [key] to type ['b]. *)

  val empty: 'b r
        (* The empty relation. *)

  val add: key -> 'b -> 'b r -> 'b r
        (* [add x y r] returns a relation containing the same bindings as
           [r], plus a binding of [x] to [y] *)

  val find: key -> 'b r -> 'b list
        (* [find x r] returns the list of current bindings of [x] in [r],
           or [] if no such binding exists. *)

  val remove: key -> 'b r -> 'b r
        (* [remove x r] returns a relation containing the same bindings
           as [r] except the bindings of [x]. Not_found is raised
           if [x] is not bound in [r]. *)

  val iter: (key -> 'b -> unit) -> 'b r -> unit
        (* [iter f r] applies [f] to all bindings in relation [r],
           discarding the results.
           [f] receives the key as first argument, and the associated value
           as second argument. The order in which the bindings are passed to
           [f] is unspecified. *)

  val iter_list: (key -> 'b list -> unit) -> 'b r -> unit
        (* [iter f r] applies [f] to all lists of bindings in relation [r],
           discarding the results.
           [f] receives the key as first argument, and the associated list
           as second argument. The order in which the bindings are passed to
           [f] is unspecified. *)
end

module Make(Ord: OrderedType): (S with type key = Ord.t)
        (* Functor building an implementation of the relation structure
           given a totally ordered type. *)
```

Figure 8: Signature du module de relations applicatives

```
(* store.mli *)
(* applicative stores *)
(* Regis Cridlig 1995 *)

type 'a store;; (* the type of stores containing values of type 'a *)

type location;; (* abstract type *)

value new_store : unit -> 'a store

  and add_location : 'a store -> 'a -> 'a store * location

  and assign_location : 'a store -> location -> 'a -> 'a store

  and remove_location : 'a store -> location -> 'a store

  and contains_location : 'a store -> location -> bool

  and val_location : 'a store -> location -> 'a

  and iter_store : ('a -> unit) -> 'a store -> unit
;;
```

Figure 9: Signature du module de stores applicatifs

```
(* automates.ml *)
(* representation ML des automates *)
(* Re'gis Cridlig 1996 *)
(* 25/7/96 *)

open List
open Transf
open Html
open ConfigType
open String

module Make (Config: ConfigType) = struct
  (* functor defining automata given configurations *)

  open Config
  open Val
  open Envir

  module Order = struct
    type t = configuration
    let compare = compare_configuration
  end

  module ConfigTable = Table.Make(Order)


  type resultat = Etat of configuration
              | Widen of configuration * sig_block list
                                   (* il faudra elargir ici *)
              | Erreur of string

  type errbloc = Bloc of act_block
             | Bloc_widen of act_block
             | Err of string

  type label = (sequence * sig_block) list
     (* signature du nouveau bloc actif a chaque fois *)

  (* un automate = un ensemble de 1-transitions indexe'es sur l'e'tat source *)

  type transition = label * resultat

  type automate = transition list ConfigTable.t
```

Figure 10: Signature du module d'automates d'ordre supérieur

5.7 Outils d'analyse de programmes $\boxed{\text{Tâche 2.7.3}}$

Divers outils d'analyse de programmes ont été développés. Les plus récents concernent l'analyse de programmes parallèles centralisés en Concurrent Pascal [Cri95] ou distribués en CML [Cri96a]. Ces implantations sont basées sur une sémantique vraiment parallèle. L'abstraction, la sémantique abstraite, l'implantation et l'interface utilisateur sont décrites avec plus de détails dans [Cri96a].

5.8 Combinaison de l'interprétation abstraite et de l'analyse des systèmes finis $\boxed{\text{Tâche 2.7.4}}$

La principale utilisation des analyses précédentes concerne l'analyse abstraite de systèmes infinis (*abstract model checking*). Comme l'analyse du comportement de systèmes infinis est indécidable en général, et, dans le cas de grands systèmes finis est trop coûteuse en temps et en mémoire, l'utilisation de l'interprétation abstraite a été explorée pour réduire la taille de l'automate à parcourir tout en conservant la garantie de fonctionnement correct, au prix d'une possibilité d'absence de réponse [Cou95a, Cri95]. Il s'agit donc de semi-vérification : la vérification des formules logiques exprimées dans le μ-calcul sur l'automate abstrait produit par l'analyseur est consistante avec la sémantique vraiment parallèle du programme mais peut se terminer dans certains cas sans pouvoir conclure [Cri95, Cri96a]. La décidabilté caractérisant les systèmes classiques de vérification de systèmes finis est donc perdue. Mais celle-ci reste théorique puisqu'il n'y a pas de limite raisonnable sur la taille mémoire ou le temps nécessaires à la résolution du problème dans ce cadre, ce qui n'est pas une situation plus favorable qu'une analyse abstraite qui termine rapidement sans pouvoir conclure, la finesse de l'analyse étant modulable par le choix d'une analyse plus fine.

Les méthodes de model-checking abstrait fournissent des moyens nouveaux d'analyse automatique du logiciel. Ces méthodes rentrent pour l'instant dans deux catégories. Une première catégorie concerne les méthodes symboliques où les propriétés du système sont approchées en utilisant des domaines abstraits. Dans ce cas, on considère des représentations approchées des ensembles d'états. Une seconde catégorie concerne le model-checking abstrait, où la sémantique d'un système de transitions infini est abstraite pour obtenir une approximation finie sur laquelle le model-checking en logique temporelle/μ-calcul peut être appliquée directement. Dans cet autre cas, on considère des représentations approchées d'ensembles de transitions. Dans [CC97] est développée une troisième possibilité d'interaction entre les méthodes d'analyse de logiciel fondées sur l'interprétation abstraite et celles fondées sur le model-checking. Il n'est fait aucune approximation des ensembles d'états ou de transitions. À la place, on effectue une analyse préliminaire ou en parallèle par interprétation abstraite sure le système d'états infini pour restreindre l'espace des états et des transitions qui devront être explorées pendant le processus de vérification. Cette technique ouvre la voie à de nouvelles méthodes d'analyse, combinant interprétation abstraite et vérification de systèmes à états finis de façon plus souple que celles précédemment utilisées.

5.9 Logique temporelle et model-checking abstrait $\boxed{\text{Tâche 2.7.4}}$

L'idée du model-checking abstrait est de valider les formules de logique modale qui caractérisent certaines propriétés comme les étreintes fatales (deadlocks), l'équité ou l'exclusion mutuelle sur les systèmes de transitions approchés obtenus par interprétation abstraite, et d'en tirer une information sur la sémantique concrète. En effet, contrairement au model-checking traditionnel, le système de transitions d'un programme n'est pas calculable en général.

On peut quand même tirer parti du fait qu'il existe d'excellents algorithmes de model-checking sur les automates. La difficulté principale est de réussir à transférer la validité d'une formule de logique temporelle suffisamment expressive de la sémantique abstraite vers la sémantique standard. Ce résultat a été obtenu pour le μ-calcul complet grâce à une interprétation abstraite qui allie à la fois une approximation supérieure et une approximation inférieure des états concrets.

Un évaluateur de formules logiques du μ-calcul est testé en sortie de l'interprète abstrait pour réaliser un outil général d'analyse de programmes parallèles.

6 Travaux effectués chez Bull

L'activité des équipes de Bull s'est orientée dans deux directions : celle de fournisseur de bibliothèque réutilisable de composants (BDD, preuve au premier ordre), et celle d'intégrateur de bibliothèques fournies par les partenaires du projet. L'intégration s'est effectuée notamment dans le cadre du projet ASPIC (anciennement ESSAIM), visant au développement d'un outil de vérification des programmes écrits en langage C, et qui sera décrit en section 6.2.

6.1 Bibliothèques fournies

6.1.1 Diagrammes de décision binaire (BDD) $\boxed{\text{Tâche 2.7.1}}$

Les diagrammes de décision binaire (BDD) sont une structure de données très simple qui permet de représenter de grosses formules booléennes efficacement. De telles formules sont utilisées notamment en vérification de matériel [Bry86], où elles représentent les valeurs de signaux circulant sur des lignes matérielles. Les variables booléennes représentent les différents signaux, et sont combinées par les opérations logiques usuelles, représentant l'action des portes logiques. Les BDD sont usuellement représentés sous la forme des TDG de Jean-Paul Billon [Bil87a], une forme légèrement plus compacte et sur laquelle la négation s'effectue en temps constant.

Les BDD connaissent de plus en plus de succès et ont vocation à être utilisés non plus exclusivement en preuve de matériel, mais aussi en analyse de programmes (section 5.2), en démonstration automatique (section 6.1.2), ou en vérification de protocoles et de systèmes réactifs (section 4.5.2). La bibliothèque de BDD développée chez Bull [BM88, Mad90] a été fournie aux partenaires du projet comme module de base en juin 1993.

6.1.2 Démonstrateur en logique du premier ordre $\boxed{\textbf{Tâche 2.7.1}}$

Il est naturel de chercher à étendre un outil tel que les BDD à un langage plus expressif. La logique du premier ordre est un candidat naturel, puisqu'il permet d'exprimer des propriétés que l'on ne peut pas exprimer en logique propositionnelle, le langage des BDD : notamment les expressions quantifiées, de la forme $\forall x \cdot P(x)$ ou $\exists x \cdot P(x)$, où x parcourt un domaine non nécessairement fini. Étendre les BDD pour traiter de la preuve de formules du premier ordre est une voie de recherche innovante suivie chez Bull à l'occasion de ce projet.

Bien que les techniques actuelles ne soient pas encore à un niveau d'avancement tel qu'elles puissent concurrencer les autres techniques de vérification formelle et d'analyse statique de programmes, de grands progrès ont été effectués. Il est envisageable d'utiliser ces techniques comme moyen logique auxiliaire de déduction, notamment dans un outil d'analyse de programmes, où les conditions booléennes calculées pourraient bénéficier d'être exprimées dans le langage plus expressif de la logique du premier ordre.

Le fait que les techniques de déduction automatique ne soient pas encore aussi efficaces que des procédures approchées, comme l'interprétation abstraite, est le prix à payer pour l'exactitude du résultat. Ce prix est nécessaire dans toutes les applications à sécurité critique ("safety-critical"), et des expériences avaient déjà été menées chez Bull, dont le but était d'assurer le fonctionnement correct (ou de détecter les dysfonctionnements) de protocoles de communication dans des couches basses de systèmes d'exploitation. Dans le cadre de la présente étude, cependant, il s'est avéré que ce prix à payer était trop fort dans les cas usuels de techniques statiques d'analyse de programmes, et que les techniques du premier ordre n'étaient pas d'importance prioritaire. Il s'ensuit que l'intégration de ces techniques de preuve n'a été effectuée dans aucun des outils développés par les différents partenaires. Son interfaçage n'a donc pas été nécessaire.

L'inclusion de techniques de preuve au premier ordre dans un outil d'analyse de programmes reste une option intéressante, mais dans un cadre plus restreint que celui de la présente étude, focalisé davantage sur un plus grand niveau d'assurance de sécurité. D'un autre côté, ces techniques de preuve seront intéressantes dans d'autres cadres. Notamment, dans une des suites du projet, le G.I.E. Dyade et notamment l'action VIP (Verified Internet Protocols), il sera nécessaire d'utiliser des techniques de preuve pour assurer la correction de protocoles cryptographiques, entre autres.

La première version du démonstrateur de théorèmes automatique du premier ordre, réalisé par Jean Goubault (Bull) dans le cadre de sa thèse [Gou93c, Gou94b], est disponible depuis novembre 1993. L'algorithme a donné lieu à dépôt de brevet (numéro PCT/FR94/01089, dépôt international du 19 septembre 1994, date de priorité du 17 septembre 1993, catégories CIB6, G06F17/50, G06F9/44).

6.2 Le projet ASPIC

Le projet ASPIC vise à la réalisation d'un outil de vérification statique des programmes écrits en langage C. Son but est de détecter les erreurs de fonctionnement de logiciels, que ce soit lors des phases de test unitaires, de validation globale, lors de la qualification ou plus tard, en phase de maintenance. Lors du portage d'applications dont la conception a été insuffisamment documentée, l'outil de vérification doit aussi permettre d'inférer suffisamment d'informations statiquement pour faciliter l'opération de portage. Finalement, ces informations doivent permettre à des outils de génération automatique de tests de produire des ensembles de tests pertinents.

Ces activités de test, de validation, qualification et maintenance sont coûteuses, il est donc crucial de développer des outils et des méthodes permettant de systématiser, voire d'automatiser les détections d'erreurs, de leurs causes, et des conditions dans lesquelles elles se produisent. L'outil ASPIC effectue cette tâche par analyse *statique* et *sémantique*.

Cette vérification est statique car elle se fait en analysant le texte des fichiers sources du programme. Elle est également sémantique car elle déduit des propriétés liées à l'exécution, au lieu de s'appuyer uniquement sur la syntaxe du langage comme beaucoup d'outils industriels existants.

Contrairement à l'activité de test dynamique qui exige l'exécution effective du programme et qui vérifie une propriété uniquement pour les cas considérés, l'analyse sémantique vérifie des propriétés pour *toutes* les exécutions possibles du programme.

Notons que ce type d'analyse ne prend pas en compte l'environnement d'exécution du programme analysé : types de réponses de l'utilisateur à des questions posées au terminal, contraintes sur les données inscrites dans les fichiers lus par l'application, etc. Ceci restreint la portée de la vérification aux fichiers sources seuls, mais permet d'un autre côté de pouvoir effectuer des vérifications alors que le véritable environnement d'exécution est indisponible ou non encore réalisé, et de fournir des garanties de sécurité de fonctionnement indépendantes de l'environnement en question.

Différents types d'erreurs peuvent être pris en compte par l'outil ASPIC :

- Les erreurs de fonctionnement *intrinsèques*, c'est-à-dire les comportements en contradiction avec la sémantique du langage : débordement de tableaux, utilisation de pointeurs ne contenant pas d'adresse valide notamment.

- Les erreurs de *conformité*, lorsque le comportement du programme est en contradiction avec les propriétés spécifiées par l'utilisateur de l'outil.

- Les non-respect des contraintes de programmation liées à la sécurité, notamment la détection de canaux cachés permettant d'accéder à des données confidentielles depuis l'extérieur du programme.

Pour accomplir ceci, une intégration de diverses techniques, venant de l'analyse statique et des méthodes formelles, est nécessaire. L'approche mathématique apporte une preuve de correction exhaustive sur l'ensemble des séquences d'exécution.

L'outil ASPIC a donné lieu récemment à un contrat entre Bull S.A. et une société extérieure.

6.2.1 Description fonctionnelle **Tâche 2.7.3**

ASPIC est un outil interactif qui prend en entrée un programme, effectue une première analyse puis répond aux requêtes de l'utilisateur.

Les programmes visés sont des programmes en ANSI-C éventuellement constitués de plusieurs fichiers, et avec un point d'entrée. Les requêtes sur le programme sont de trois types :

- localiser les accès à une variable en écriture ou en lecture dans tout le programme;
- calculer les conditions sous lesquelles une instruction donnée est exécutée, calculer la valeur d'une variable en un point donné;
- repérer les possibilités de débordement d'une zone mémoire dans tout le programme.

L'outil ASPIC est conçu pour fonctionner en deux modes : le mode *réponse directe* et le mode réponse en *analyse par cas.* L'utilisateur fournit tout d'abord le programme, qui est prédigéré par une phase de traitements préliminaires. Il peut ensuite poser des questions entrant dans les catégories de requêtes décrites plus haut : en mode réponse directe, l'outil ASPIC présente les informations qu'il a réussi à calculer en réponse à la requête, en tenant compte de tous les chemins d'exécution possibles. L'utilisateur a ensuite la possibilité d'affiner sa requête en précisant les valeurs d'entrée de certains des paramètres, dans le but soit de rendre plus précise l'analyse, soit de rendre plus lisible une réponse trop complexe à déchiffrer : c'est le mode analyse par cas.

En mode analyse par cas, ASPIC génère aléatoirement des valeurs pour les variables d'entrée et propose un ensemble de résultats partiels correspondant aux valeurs particulières des variables d'entrée. Le calcul de ces valeurs s'appuie sur une analyse data-flow d'une part, et sur les résultats obtenus en mode direct d'autre part, de manière à être le plus pertinent possible pour la requête.

L'architecture d'ASPIC est donnée en figure 11. Les modules sont répartis dans trois parties distinctes : les traitements préliminaires; le moteur d'analyse sémantique, qui est activé à chacune des requêtes; et la génération de cas d'analyse qui est déclenchée à la demande de l'utilisateur.

Nous décrivons ci-après les diverses techniques employées dans les sous-sections suivantes :

6.2.2 Traitements préliminaires

Avant toute interrogation sur le programme à analyser, des traitements préliminaires sont effectués sur l'ensemble des sources de ce programme. Ces traitements ont pour objet de transformer le code C en une représentation interne plus facile à manipuler par les différentes techniques d'analyse. Ils permettent également de calculer des informations sémantiques utilisées ultérieurement par ces techniques.

La représentation interne du programme est obtenue par des techniques d'analyse lexicale et syntaxique classiques [WM94, ASU86]. Une fois cette représentation obtenue, les informations suivantes sont calculées :

1. le graphe de contrôle interprocédural,
2. le graphe d'appel des procédures et des fonctions,
3. les alias en chaque point du programme, c'est-à-dire l'ensemble des symboles permettant d'accéder à une même adresse mémoire. Cet ensemble est obtenu par l'analyse des pointeurs [Lan92] utilisés dans les différentes procédures et lors du passage des paramètres à l'appel de ces procédures;

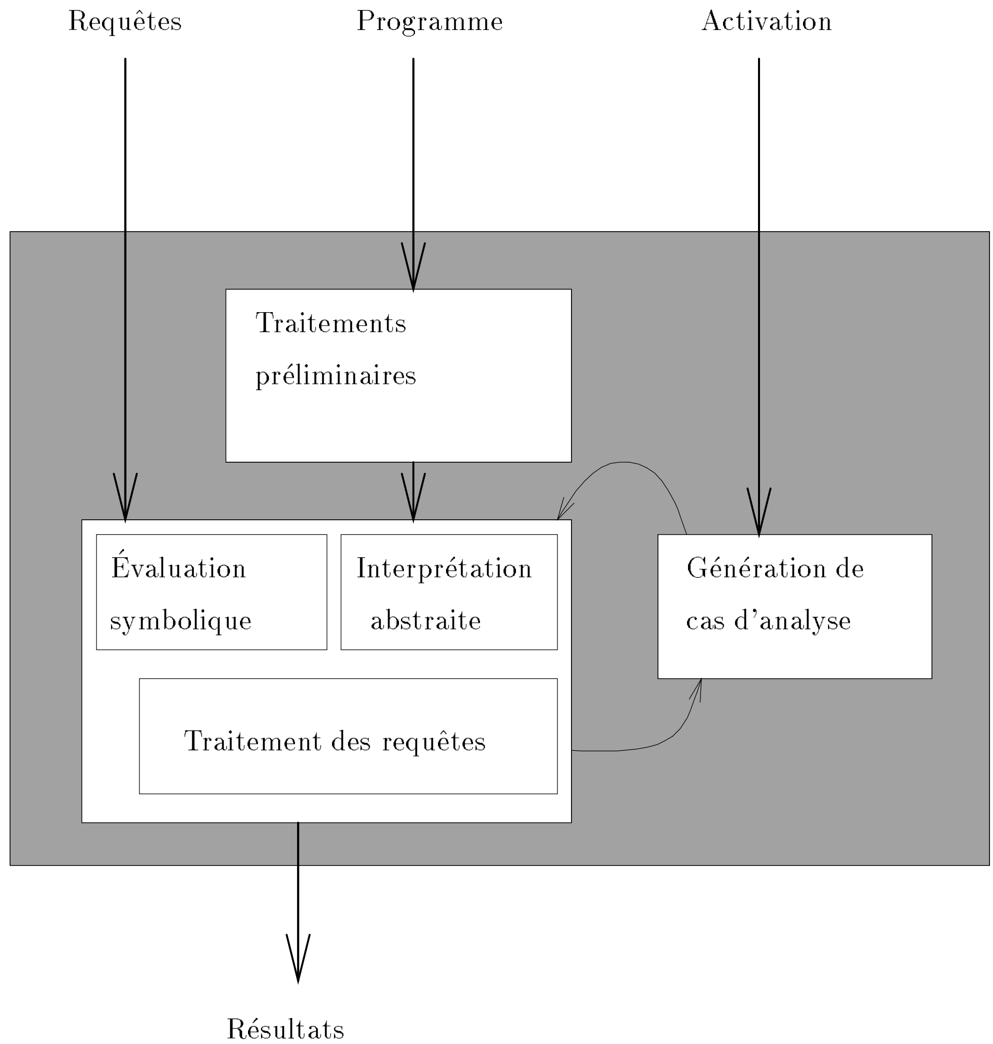


Figure 11: Architecture d'ASPIC

4. la relation *use-def* [HS94], c'est-à-dire, por chaque utilisation du contenu d'une adresse mémoire donnée, l'ensemble des instructions du programme qui définissent cette adresse mémoire et dont la définition atteint l'utilisation considérée;

5. les informations préliminaires spécifiques à chacune des deux techniques d'analyse sémantique (voir section 6.2.3).

6.2.3 Gestion des requêtes — **Tâche 2.7.3, 2.7.4**

L'approche d'ASPIC est exhaustive, elle couvre l'ensemble des exécutions du programme. L'analyse effectuée utilise des approximations de l'exécution qui garantissent toutes une approche *safe* : l'analyse ne laisse échapper aucun cas d'erreur, mais peut en signaler plus qu'il n'y en a.

Cette approche permet de traiter une grande partie des constructions du langage C. Cependant, plus les structures manipulées par le programme C sont complexes, plus la perte de précision est importante. En particulier, les requêtes sur les parties de programme comportant des pointeurs à plusieurs niveaux ou des procédures récursives n'auront en général pas de réponse précise.

Localisation des accès en lecture ou écriture des variables La réponse à une requête de localisation est donnée comme un sur-ensemble de l'ensemble des points du programme où la variable désignée est lue ou écrite. Un cas intéressant est celui où l'ensemble calculé par ASPIC est vide, auquel cas on est sûr qu'aucun accès n'est fait à cette variable.

Cette analyse tient compte des conditions d'exécution liées à chaque chemin d'exécution possible du programme, comme suit.

Condition d'exécution et demande de valeur en un point La réponse à une requête de conditions sous lesquelles une instruction est exécutée est donnée sous forme d'une expression logique. L'analyse du programme est effectuée *symboliquement*, en paramétrant les valeurs calculées des variables et les conditions booléennes d'exécution de chaque branche du programme par des variables symboliques représentant les valeurs des variables d'entrée du programme. On appelle variable d'entrée tous les paramètres du programme principal, les valeurs lues depuis des fichiers ou depuis la console, et les variables non définies dans le module à analyser.

Plus précisément, l'analyse sémantique permettant d'aboutir à ce genre de réponses repose sur deux techniques d'interprétation abstraite, l'*évaluation symbolique* [CR81, Cou78, Deh90], et des techniques plus simples, non symboliques comme l'*analyse d'intervalles* [CC77a, Bou92]. Ces deux techniques ont été retenues pour leur complémentarité dans l'analyse des programmes impératifs. En effet, l'évaluation symbolique s'avère performante dans l'analyse des structures conditionnelles mais perd beaucoup d'information sur les variables modifiées dans les boucles. En revanche, l'analyse d'intervalles permet de conserver plus d'informations dans les boucles mais, étant une interprétation abstraite non distributive, perd rapidement en précision sur les variables affectées par les branchements conditionnels :

- L'évaluation symbolique est une interprétation non standard qui simule l'exécution du programme en l'absence de valeurs d'entrée. Elle associe aux variables du programme des *valeurs symboliques*. Ces valeurs sont des expressions en fonction des valeurs des variables d'entrée, comme l'expression $1 + y_0$, par exemple, où y_0 est une variable d'entrée.

  Chaque valeur symbolique est *gardée* par une expression symbolique qui constitue une condition nécessaire pour atteindre la valeur symbolique au point de programme concerné. Cette garde représente donc un sur-ensemble des chemins d'exécution permettant d'obtenir ma valeur symbolique considérée.

  À tout point de programme est ainsi associé une formule logique formée à partir des variables symboliques, et représentant une condition nécessaire pour que le point de programme soit atteint au cours d'une exécution arbitraire. Cette *condition d'exécution* de l'instruction considérée est, en particulier, la disjonction des gardes associées aux valeurs symboliques d'une variable quelconque de la mémoire en cours.

  Un cas trivial mais particulièrement intéssant est celui ou l'expression logique retournée est FAUX, ce qui signifie qu'il est impossible de passer par l'instruction visée; autrement dit, le code situé à ce point de programme est du *code mort*, qui ne sera jamais exécuté. Si cela n'est pas intentionnel, cela indique en général une erreur dans l'écriture des tests rencontrés plus hauts dans le programme.

  Moins trivialement, une condition d'exécution de base est un prédicat exprimé dans l'un des treillis d'interprétation abstraite de base, par exemple :

  $$u_0 + v_0 > 10 \wedge v_0 \leq 15$$

  Ici, u_0 et v_0 sont des variables symboliques dénotant les valeurs des variables u et v d'un sous-programme donné, à l'entrée du sous-programme. Ce sont des *variables d'entrée*, au sens décrit plus haut.

  Les valeurs symboliques de variables sont alors données par des listes finies de conditions gardées, paramétrées par les variables d'entrée, par exemple :

  $$x = \begin{cases} 1 + y_0 & \text{si } u_0 + v_0 > 10 \\ 3 - y_0 & \text{si } u_0 + v_0 < 10 \end{cases}$$

- La propagation d'intervalles, et plus généralement les techniques non symboliques non distributives, d'interprétation abstraite sont utilisées pour renforcer les résultats de l'exécution symbolique. Celles utilisées dans le projet ASPIC consistent à associer à chaque point du programme et à chaque variable un ensemble de valeurs qui contient toutes les valeurs que pourra jamais prendre cette variable en ce point. La formalisation des ensembles de valeurs possibles est opérée via la définition d'un *domaine abstrait* de valeurs, qui peut être défini par des intervalles de valeurs (cf. section 5.5), des congruences [Gra91a, Gra91b, Mas93a, Mas93b], des relations affines entre variables [Ka76] (voir aussi en section 5.5).

Dans le cas du projet ASPIC, l'effort porte davantage sur la détection de dysfonctionnements que sur l'optimisation ou la parallélisation de programmes. Dans ce cadre, la détection de congruences ou de relations affines entre variables apporte moins que la propagation d'intervalles de valeurs. En effet, les menaces sur le fonctionnement du programme sont souvent exprimées par l'association d'une valeur *dangereuse* à une variable *critique*. L'analyse d'intervalles retourne pour chaque variable numérique, disons x, une expression de la forme $x \in [7, 25]$, signifiant qu'au point de programme considéré la variable x prend nécessairement une valeur comprise entre 7 et 25 (à cause de la perte de précision due à l'analyse, il se peut que les valeurs réelles de x ne parcourent pas tout l'intervalle $[7, 25]$). Si les valeurs dangereuses pour x sont les entiers négatifs ou nuls, le résultat $x \in [7, 25]$ assure qu'il n'y a aucun risque de dysfonctionnement dû à la variable critique x en ce point. Sinon, l'analyse doit être affinée en mode analyse par cas.

Il est aussi envisagé d'étendre l'analyse par intervalles en utilisant le treillis des polyèdres convexes de l'IMAG [CH78, Hal79] (cf. section 4.2.1).

Débordement d'une zone mémoire La réponse donnée à une requête de détection des débordements de zones mémoire est donnée comme un sous-ensemble de tous les points de programme auxquels le débordement de la zone est possible. Les techniques utilisées sont les mêmes qu'aux paragraphes précédents.

6.2.4 Génération de cas d'analyse

Au vu des résultats d'une requête, l'utilisateur peut demander à préciser ces résultats en activant la *génération de cas d'analyse*. Différents cas à analyser sont alors générés suivant un critère de test de type *all-uses* [FL88] : étant donnée une variable en un point donné du programme, on recherche toutes les définitions de cette variable susceptibles d'être encore valides à ce point.

Le processus de génération des cas d'analyse est organisé en trois parties : le calcul de la relation *use-def* sur l'ensemble du programme en traitement préliminaire, la caractérisation des différents cas d'analyse possible associés à une requête donnée et, enfin, la génération aléatoire des valeurs pour tout ou partie des variables d'entrée dans chaque cas d'analyse obtenu.

Pour chaque cas produit, les deux techniques d'analyse sémantique sont réactivées en prenant en compte les valeurs d'entrée générées. La nouvelle analyse effectuée est donc restreinte à un sous-ensemble des chemins d'exécution associés à la requête.

6.2.5 Autres modules **Tâche 2.7.4**

Le *traitement des requêtes* a pour objet de récupérer dans les mémoires symboliques et abstraites, issues des deux techniques d'analyse, les résultats concernant la requête en cours. Il simplifie au mieux la conjonction de ces résultats et les présente de façon claire et concise à l'utilisateur.

Un module contenant des *outils de simplification* des expressions est utilisé par l'ensemble des modules de l'analyseur. Actuellement, il est composé de deux outils :

- Simplification des expressions logiques. Cette simplification passe par la forme canonique des BDD, et s'appuie sur la bibliothèque de BDD du centre de recherches de Bull (cf. section 6.1.1).
- Simplification des systèmes de contraintes linéaires. Cette simplification est basée sur des techniques dérivées de l'algorithme de Chernikova [Lev92] (cf. section 4.2.1).

6.2.6 Exemples

Nous donnons ici deux exemples illustrant les différentes requêtes et les deux modes proposés par ASPIC.

```
(0)   int exemple1 (int x)
      {
(1)   int u, v, w, acces, tmp, tab[11]; double res;
(2)   u = 1; v = 2; w = 3; tmp = 0;
(3)   if (x>0)
(4)       { acces = 1; tmp = u+2*v+w; tab[1] = tab[tmp]; }
      else
(5)       acces = 0;
(6)   res = 0;
(7)   while (tmp>0)
(8)       { res = res+tmp; tmp = tmp-1; }
(9)   tab[2] = tab[res];
(10)  if (tmp<=0 || acces==1)
(11)      res = cos (sqrt (res));
      else
(12)      res = -12; /* code mort */
(13)  return x;
      }
```

Figure 12: Exemple de programme

On examine le programme de la figure 12. Voici quelques requêtes :

1. Valeur des variable `tmp`, `acces`, `res` au point (13) :

   Réponse en (13) : `tmp`$\in [0,0]$
   Réponse en (13) : `acces`$= 0$ si $(x_0 > 0)$ et `acces`$= 1$ si $(x_0 \leq 0)$
   Réponse en (13) : `res`$\in [-1,1]$

La valeur de la variable acces est donnée en fonction d'une variable d'entrée x_0, dénotant la valeur initiale (en (1)) du paramètre x. Chaque nouvelle valeur inconnue prise par la variable x est numérotée, par exemple si l'on rencontre une instruction de la forme :

```
fscanf(file, "%d", &x);
```

Il est donc possible d'obtenir des expressions de la forme $x_0 + y_7$.

2. Valeur de la variable res en (9) :

   Réponse en (9) : res$\in [0, +\infty[$

   Ceci signifie que l'on peut être sûr que la valeur de la variable res est positive à ce point du programme, et que l'on peut donc calculer sa racine carrée. On remarquera que la borne $+\infty$ vient d'une approximation effectuée lors du passage dans la boucle (7)–(8).

3. Condition d'exécution des instructions (12), (8) et (5) :

   Réponse en (12) : FAUX

   Le code en question est donc mort, il ne sera jamais exécuté (comme indiqué dans le commentaire).

   Réponse en (8) : condition indéterminée

   La condition n'a pas pu être déterminée en fonction des valeurs des variables d'entrée. Le résultat retourné est en fait VRAI, mais ceci n'est qu'une condition nécessaire d'exécution et non suffisante, et n'est donc pas informative.

   Réponse en (5) : x≤ 0

   On ne passera donc en (5) que si x est négative ou nulle. Ceci peut sembler évident sur cet exemple, mais l'est beaucoup moins sur des codes plus gros.

4. Localisation de tmp en lecture et en écriture :

   Écriture : (4), (8)
   Lecture : (4), (8), (10)

   La localisation n'est pas uniquement syntaxique, elle identifie aussi les accès au travers de pointeurs. Par exemple, si l'on a la suite d'instructions suivante :

```
p = &x; *p = 12;
```

l'analyseur détectera correctement l'accès en écriture à x.

```
(0)  int exemple2 (int x, int a)
     {
(1)  int b, result;
(2)  b = 0;
(3)  while (x<=10)
       {
(4)      if (a==0)
(5)          b = 1;
         else
(6)          b = 1000;
(7)      x += 1;
       }
(8)  result = 5*b+100;
(9)  return result;
     }
```

Figure 13: Exemple de programme, bis

5. Débordement du tableau `tab` :

   Ligne(s) (9)

   La ligne (4) n'apparaît pas dans la liste car en cet endroit, on est assuré que le débordement est impossible.

Examinons maintenant le programme de la figure 13. Dans ce programme, si l'on demande la valeur de la variable `result` en ligne (8), on obtient en analyse directe :
`result`$\in [100, 5100]$
L'imprécision du résultat est due aux deux raisons suivantes :

- L'évaluation symbolique ne conserve pas les valeurs 1 et 1000 affectées à `b` à l'intérieur de la boucle.
- L'interprétation abstraite fondée sur l'analyse d'intervalles calcule l'union des intervalles $[0, 0]$, $[1, 1]$, et $[1000, 1000]$ associés à `b` en ligne (7), et retourne le plus petit intervalle contenant cette union, soit $[0, 1000]$. La variable `result` reçoit alors la valeur abstraite $[100, 5100]$ en (8).

L'activation de la génération de cas d'analyse déterminera sur ce cas les éléments suivants :

1. La valeur de `result` dépend de la variable `b`.
2. Les dernières définitions de `b` encore valables en (8) sont celles associées aux instructions (2), (5) et (6).

La génération de cas va forcer trois analyses restreintes qui permettent d'obtenir des résultats plus précis. Une première analyse considère uniquement le chemin d'exécution qui passe par l'instruction (2) sans rentrer dans la boucle `while` en générant, par exemple, les valeurs 25 et 0 pour les variables d'entrée `x` et `a`. L'analyse donne alors `result`$\in [100, 100]$, autrement dit `result` vaut 100 en (9). Puis une second analyse est forcée qui considère le chemin d'exécution passant par l'instruction (5), en générant par exemple les valeurs 5 et 0 pour `x` et `a`. Cette analyse donne alors `result`$\in [105, 105]$. Enfin, une troisième analyse considère le chemin d'exécution qui passe par l'instruction (6), en générant par exemple les valeurs 10 et 98 pour `x` et `a`. L'analyse donne alors `result`$\in [5100, 5100]$.

6.2.7 Apport

Tâche 2.7.3

ASPIC est un projet de transfert de technologie, qui intègre les différentes techniques et technologies des trois partenaires de l'étude. L'objectif est de montrer d'une part la complémentarité de ces techniques, et d'autre part d'industrialiser ces techniques afin d'obtenir un outil capable d'analyser des programmes réels.

L'originalité de l'approche réside dans la mise en commun des différents résultats obtenus, et dans l'analyse par cas visant à rendre plus clairs ou plus précis ces résultats. Une étape ultérieure sera de mêler plus étroitement les différentes analyses effectuées afin d'améliorer la précision des résultats.

Pour l'instant, ASPIC est un prototype qui n'intègre qu'une partie des techniques mises à disposition par les autres partenaires de l'étude. Conçu comme une plateforme, l'outil ASPIC doit permettre à l'avenir d'introduire de nouvelles techniques et de les associer pour trouver un optimum dans le compromis entre temps d'exécution, place mémoire consommée et précision des résultats. Cette flexibilité permettra de s'adapter aux programmes à vérifier qui, selon leur nature, ne nécessiteront pas l'emploi des mêmes techniques.

Une maquette a été réalisée en EIFFEL au début de l'année 1996, permettant d'utiliser les techniques retenues dans un contexte intraprocédural avec des pointeurs à un seul niveau. Une première version de l'outil prévue pour début 1997 permettra une analyse interprocédurale et acceptera des pointeurs à plusieurs niveaux ainsi que les procédures récursives. Quelques restrictions persisteront encore sur les programmes analysés, notamment l'interdiction des *cast*, de l'arithmétique sur les pointeurs, et des boucles non naturelles, c'est-à-dire à plusieurs points d'entrée.

Le but final est de produire un outil qui accepte l'ensemble du langage ANSI-C, et d'évaluer les différentes techniques d'analyse en complexité spatiale et temporelle, et en précision. Ces paramètres dépendent de la longueur du code examiné, mais surtout de l'imbrication des structures de contrôle, du niveau des pointeurs et de l'aspect récursif de l'appel des fonctions ou des structures de données.

Un certain nombre d'évolutions à plusieurs niveaux sont d'ores et déjà souhaitables pour rendre l'outil ASPIC véritablement opérationnel dans un contexte industriel :

1. améliorer la précision des techniques et l'interaction entre celles-ci;
2. permettre à l'utilisateur d'entrer des propriétés sur le programme qui lui semblent évidentes, comme des invariants de boucles, des préconditions ou des postconditions

sur certaines des procédures;

3. adjoindre de nouveaux outils : démonstrateur de théorèmes (cf. section 6.1.2), outils de simplification, etc.;

4. étendre les fonctionnalités : augmenter le pouvoir d'expression du langage de requêtes, détecter les canaux cachés, générer automatiquement des jeux de tests, etc.

L'architecture de l'outil, et notamment la forme de la représentation interne utilisée, devrait permettre d'adapter ASPIC rapidement à d'autres langages. Les langages impératifs comme COBOL ou FORTRAN ne devraient pas poser de difficultés majeures, de même que les langages spécialisés dédiés aux applications sécuritaires qui sont généralement beaucoup moins permissifs que le langage C.

7 Conclusion

Le but de cette étude était de mettre en commun les compétences en matière de méthodes formelles, d'analyse sémantique et de vérification formelle de trois laboratoires de recherche, celui de Bull S.A. (Les Clayes-sous-Bois), de l'IMAG (Grenoble) et du LIENS (Paris), dans le but d'opérer une véritable synergie tant des efforts de recherche que des applications industrielles.

Malgré quelques difficultés dans l'organisation du projet, dû à une révision inopinée de la politique de financement du ministère, et qui a forcé chaque laboratoire à revoir son organisation et ses moyens de financement, les résultats obtenus sont dans la ligne de ce qui était initialement prévu. Les engagements des trois équipes de recherche ont été tenus, sinon à la lettre, du moins dans leurs aspects fondamentaux : réalisations techniques, coopération, collaboration des équipes, et intégration des outils de chaque équipe selon les besoins propres de chacun des partenaires.

Les trois équipes continueront leur collaboration dans les domaines respectifs d'intérêt qui ont été les leurs dans cette étude. Les perspectives de recherches et de développements sont en effet très nombreuses aussi bien dans les domaines de vérification de systèmes finis ou infinis d'états, que de l'analyse de programmes, dans des buts d'optimisation ou de détection de défauts de sécurité notamment.

References

[ABM93] Y. Afek, G. Brown, and M. Meritt. Lazy caching. *ACM Transactions on Programming Languages and Systems*, 15(1), 1993.

[ASU86] A.V. Aho, R. Sethi, and J.D. Ullman *Compilers: Principles, Techniques and Tools.* Addison-Wesley, 1986.

[Ake78] S. B. Akers. Binary decision diagrams. *IEEE Transactions on Computers*, C-27(6), 1978.

[ACD90] R. Alur, C. Courcoubetis, et D. Dill. Model checking of real-time systems. In *Fifth IEEE Symposium on Logic in Computer Science*, 1990.

[ACH+95] R. Alur, C. Courcoubetis, N. Halbwachs, T. Henzinger, P. Ho, X. Nicollin, A. Olivero, J. Sifakis, and S. Yovine. The algorithmic analysis of hybrid systems. *Theoretical Computer Science B*, 138:3–34, janvier 1995.

[AD90] R. Alur and D. Dill. Automata for modeling real-time systems. In *ICALP'90*, 1990.

[Anc86] F. Anceau, Design methodology for large custom processors, in Proceedings of *the 1986 ESSIR Conference*, Delft, USA, 1986.

[ABM+89] F. Anceau, C. Berthet, J. C. Madre, O. Coudert, J. P. Billon, La preuve formelle en milieu industriel, in *TSI*, Vol. 8, N°6, 1989.

[Baz92] P. Bazet. Pattern-matching en priorité spécifique sur des types avec constructeurs commutatifs. Rapport de recherche, INRIA/Université d'Orsay, septembre 1992.

[BCM90] C. Berthet, O. Coudert, J. C. Madre, New ideas on symbolic manipulations of finite state machines, in Proceedings of *ICCD'90*, Cambridge MA, USA, September 1990.

** Best Paper Award **

[Bil87a] J. P. Billon, Perfect normal forms for discrete functions, *BULL Research Report* N°87019, June 1987.

[Bil87b] J. P. Billon, Symbolic execution of discrete programs, *BULL Research Report* N°87039, September 1987.

[Bil91] Billon J. P., A new approach of theorem proving for non clausal first order logic with equality based on generalized Shanon's decomposition principle, *BULL Research Report*, December 1991.

[BM88] J. P. Billon, J. C. Madre, Original concepts of PRIAM, an industrial tool for efficient formal verification of combinational circuits, in *The Fusion of Hardware Design and Verification*, G. J. Milne Editor, North-Holland, pp. 487–501, July 1988.

[BFH+92] A. Bouajjani, J. C. Fernandez, N. Halbwachs, P. Raymond, et C. Ratel. Minimal state graph generation. *To appear in Science of Computer Programming*, 1992.

[BJ 82] Bjørner, D., Jones, C.B., *Formal Specification and Software Development*, Prentice-Hall, 1982.

[Bol91] Bolignano D., A treatment of equality for quantifier-free formulae, *BULL Research Report*, December 1991.

[BD 92] Bolignano, D., Debabi, M.,*On the foundations of the RAISE specification language semantics*, *BULL Research Report* (April 1992).

[Bou90] F. Bourdoncle. Interprocedural abstract interpretation of block structured languages with nested procedures, aliasing and recursivity. In P. Deransart and Małuszyński, editors, *Proceedings of the International Workshop PLILP'90, Programming Language Implementation and Logic Programming*, Linköping, Sweden, Lecture Notes in Computer Science 456, pages 307–323. Springer-Verlag, New York, 20–22 août 1990.

[Bou92] F. Bourdoncle. Sémantiques des langages impératifs d'ordre supérieur et interprétation abstraite. Thèse de l'École Polytechnique, novembre 1992.

[Bry86] R. E. Bryant. Graph-based algorithms for Boolean function manipulation. *IEEE Transactions on Computers*, C-35(8), 1986.

[BCM+89] J.R. Burch, E.M. Clarke, K.L. McMillan, D.L. Dill, et J. Hwang. *Symbolic Model Checking:* 10^{20} *states and beyond.* Rapport de Recherche, Carnegie Mellon University, 1989.

[BHA86] G.L. Burn, C. Hankin, et S. Abramsky. Strictness analysis for higher-order functions. *Science of Computer Programming* 7, pp. 249–278, 1986.

[Cha87] D.R. Chase. An improvement to bottom-up tree pattern matching. In *Proceedings of the 14th Annual ACM Symposium on PoPL*, pp.168–177. ACM, 1987.

[Che65] N. V. Chernikova. Algorithm for finding a general formula for the non-negative solution of a system of linear inequalities. *USSR Computational Mathematics and Mathematical Physics*, 5(2), pp.334-337, 1965.

[Che68] N. V. Chernikova. Algorithm for discovering the set of all solutions of a linear programming problem. *U.S.S.R. Computational Mathematics and Mathematical Physics*, 8(6):282–293, 1968.

[CES86] E. M. Clarke, E. A. Emerson, et A. P. Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. *TOPLAS*, 8(2), 1986.

[CR81] L.A. Clarke et D.J. Richardson. Symbolic evaluation methods for program analysis. In S.S. Muchnick et N.D. Jones, eds., *Program Flow Analysis: Theory and Applications*, Software Series, chapitre 9, 264–300. Prentice-Hall, 1981.

[CBM89a] O. Coudert, C. Berthet, J. C. Madre, Verification of synchronous sequential machines based on symbolic execution, in *Lecture Notes in Computer Science: Automatic Verification Methods for Finite State Systems*, Volume 407, J. Sifakis Editor, Springer-Verlag, pp. 365–373, June 1989.

[CBM89b] O. Coudert, C. Berthet, J. C. Madre, Verification of sequential machines using Boolean functional vectors, in *Formal VLSI Correctness Verification*, L. J. M. Claesen Editor, North-Holland, pp. 179–196, November 1989.

[CBM90] O. Coudert, C. Berthet, J. C. Madre, Symbolic manipulations for the verification of sequential machines, in Proceedings of *1st EDAC*, Glasgow, UK, March 1990.

[CM90a] O. Coudert, J. C. Madre, Verifying temporal properties of sequential machines without building their state diagrams, in Proceedings of *CAV'90*, E. M. Clarke and R. P Kurshan Editors, DIMACS Series, pp. 75–84, June 1990.

[CM90b] O. Coudert, J. C. Madre, A unified framework for the formal verification of sequential circuits, in Proceedings of *ICCAD'90*, Santa Clara CA, USA, November 1990.

** Distinguished Paper **

[CM91a] O. Coudert, J. C. Madre, Symbolic computation of the valid states of a sequential machine: Algorithms and discussion, in Proceedings of *International Worshop on Formal Methods in VLSI Design*, Miami FL, USA, January 1991.

[Cou91] O. Coudert, *SIAM : Une Boîte à Outils pour la Preuve Formelle de Systèmes Séquentiels*, Thèse de troisième cycle (PhD. Thesis), Ecole Nationale Supérieure des Télécommunications, Paris, France, October 1991.

[CM91b] O. Coudert, J. C. Madre, A new method to compute prime and essential prime implicants of Boolean functions, *BULL Research Report*, N°91028, October 1991.

[CM91c] O. Coudert, J. C. Madre, Implicit and incremental computation of prime and essential prime implicants of Boolean functions, *BULL Research Report*, N°91033, November 1991.

[CM91d] O. Coudert, J. C. Madre, Symbolic prime computation of multiple-output Boolean functions, *BULL Research Report* N°91034, November 1991.

[CM92a] O. Coudert, J. C. Madre, A new method to compute prime and essential prime implicants of Boolean functions, in *Advanced research in VLSI and Parallel Systems*, T. Knight and J. Savage Editors, The MIT Press, pp. 113–128, March 1992.

[CM92b] O. Coudert, J. C. Madre, Implicit and incremental computation of primes and essential primes of Boolean functions, in Proceedings of *29th DAC*, Anaheim CA, USA, June 1992.

[CM92c] O. Coudert, J. C. Madre, A new implicit DAG based prime and essential prime computation technique, in Proceedings of *International Symposium on Information Sciences*, Fukuoka, Japan, July 1992.

[CM92d] O. Coudert, J. C. Madre, The implicit set paradigm: A new approach to finite state system verification, submited to the *Formal Methods in System Design* Journal, E. M. Clarke Editor, Kluwer Academic Publishers, 1992.

[CM92e] O. Coudert, J. C. Madre, Towards an interactive fault tree analyser, submited to *IASTED Conference on Reliability, Quality Control, and Risk Assessment*, Washington D.C., USA, November 1992.

[CM93] O. Coudert, J. C. Madre, Fault tree analysis: 10^{20} prime implicants and beyond, submited to *Annual Symposium on reliability*, Atlanta NC, USA, January 1993.

[CMT93] O. Coudert, J.-C. Madre, and H. Touati. TiGeR version 1.0 user guide. Technical report, Digital Paris Research Laboratory, novembre 1993.

[CM] O. Coudert, J. C. Madre, Une approche intentionnelle du calcul des implicants premiers et essentiel des fonctions booléennes, submited to *RAIRO*.

[CC76] P. Cousot and R. Cousot. Static determination of dynamic properties of programs. In *Proceedings of the 2^{nd} International Symposium on Programming*, pages 106–130. Dunod, Paris, 1976.

[CC77a] P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Conference Record of the 4^{th} ACM Symposium on Principles of Programming Languages*, pages 238–252, Los Angeles, California, U.S.A., 1977.

[CC77b] P. Cousot and R. Cousot. Automatic synthesis of optimal invariant assertions : mathematical foundations. In *ACM Symposium on Artificial Intelligence & Programming Languages*, Rochester, New York, U.S.A., SIGPLAN Notices 12(8):1–12, 1977.

[CC77c] P. Cousot and R. Cousot. Static determination of dynamic properties of generalized type unions. In *ACM Symposium on Language Design for Reliable Software*, Raleigh, N. C., U.S.A., SIGPLAN Notices 12(3):77–94, 1977.

[CC77d] P. Cousot and R. Cousot. Static determination of dynamic properties of recursive procedures. In E. J. Neuhold, editor, *IFIP Conference on Formal Description of Programming Concepts, St-Adrews, N. B., Canada*, pages 237–277. North-Holland Pub. Co., Amsterdam, 1977.

[Cou78] P. Cousot. *Méthodes itératives de construction et d'approximation de points fixes d'opérateurs monotones sur un treillis, analyse sémantique de programmes.* Thèse d'État es sciences mathématiques, Université scientifique et médicale de Grenoble, Grenoble, France, 21 mars 1978.

[CH78] P. Cousot and N. Halbwachs. Automatic discovery of linear restraints among variables of a program. In *Conference Record of the 5^{th} ACM Symposium on Principles of Programming Languages*, pages 84–97, Tucson, Arizona, U.S.A., 1978.

[CC79] P. Cousot and R. Cousot. Systematic design of program analysis frameworks. In *Conference Record of the 6^{th} ACM Symposium on Principles of Programming Languages*, pages 269–282, San Antonio, Texas, U.S.A., 1979.

[CC80] P. Cousot and R. Cousot. Semantic analysis of communicating sequential processes. In de Bakker J. W. and J. van Leeuwen, editors, *Seventh International Colloquium on Automata, Languages and Programming*, Lecture Notes in Computer Science 85, pages 119–133. Springer-Verlag, New York, juillet 1980.

[Cou81] P. Cousot. Semantic foundations of program analysis. In S. S. Muchnick and N. D. Jones, editors, *Program Flow Analysis: Theory and Applications*, chapter 10, pages 303–342. Prentice-Hall, Inc., Englewood Cliffs, 1981.

[CC84] P. Cousot and R. Cousot. Invariance proof methods and analysis techniques for parallel programs. In A. W. Biermann, G. Guiho, and Y. Kodratoff, editors, *Automatic Program Construction Techniques*, chapter 12, pages 243–271. Macmillan, New York, 1984.

[CC92a] P. Cousot and R. Cousot. Abstract interpretation and application to logic programs. *Journal of Logic Programming*, 13(2–3), 1992. A paraître.

[CC92b] P. Cousot et R. Cousot. Comparing the Galois connection and widening/narrowing approaches to abstract interpretation, papier invité. Dans M. Bruynooghe et M. Wirsing, éditeurs, *Proceedings of the International Workshop Programming Language Implementation and Logic Programming, PLILP '92*, Louvain, Belgique, 13–17 août 1992, Lecture Notes in Computer Science 631, pages 269–295. Springer-Verlag, Berlin, Allemagne, 1992.

[CC92c] P. Cousot et R. Cousot. Constructing a hierarchy of semantics of functional programs by abstract interpretation, conférence invitée. Dans *Workshop on Static Analysis, WSA '92*, Bordeaux, France, 23–25 septembre 1992.

[CC92d] P. Cousot et R. Cousot. Inductive definitions, semantics and abstract interpretation. Dans *Conference Record of the Ninthteenth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 83–94, Albuquerque, Nouveau Mexique, 1992. ACM Press, New York, New York, USA.

[CC93a] P. Cousot et R. Cousot. Abstract interpretation of parallel programs, conférence invitée. Dans *Int. Kolloquium, Sonderforschungsbereich 124, VLSI – Entwurfsmethoden und Parallelität*, Sarrebrücken, Allemagne, 2–3 septembre 1993.

[CC93b] P. Cousot et R. Cousot. Galois connection based abstract interpretations for strictness analysis, papier invité. Dans D. Bjø rner, M. Broy, et I.V. Pottosin, éditeurs, *Proceedings of the International Conference on Formal Methods in Programming and their Applications*, Academgorodok, Novosibirsk, Russie, Lecture Notes in Computer Science 735, pages 98–127. Springer-Verlag, Berlin, Allemagne, juin 28–juillet 2, 1993.

[CC93c] P. Cousot et R. Cousot. Galois connections and abstract interpretation, conférence invitée. Dans *Mathematics of Programming Workshop on Galois Connections*, Universiteit Utrecht, Pays-Bas, 13–14 septembre 1993.

[CC93d] P. Cousot et R. Cousot. Inductive definitions of semantics and connections between semantics as used in abstract interpretation, conférence invitée. Dans *MASK Meeting*, Koblenz, Allemagne, 6–8 octobre 1993.

[CC94] P. Cousot et R. Cousot. Higher-order abstract interpretation (and application to comportment analysis generalizing strictness, termination, projection and PER analysis of functional languages), papier invité. Dans *Proceedings of the 1994 International Conference on Computer Languages*, Toulouse, France, pages 95–112. IEEE Computer Society Press, Los Alamitos, Californie, USA, 16–19 mai 1994.

[Cou95a] P. Cousot. Abstract model checking, conférence invitée. Dans *Seventh International Conference on Computer Aided Verification, CAV '95*, Liège, Belgique, 5 juillet 1995.

[Cou95b] P. Cousot. Combining bottom-up and top-down in abstract interpretation of logic languages, conférence invitée. Dans *Special Workshop on Abstract Interpretation of Logic Languages*, Eilat, Israël, juin 18–19 1995.

[Cou95c] P. Cousot. Completeness in abstract interpretation, conférence invitée. Dans M.I. Sessa et M. Alpuente, éditeurs, *Proceedings of the GULP-PRODE '95 Joint Conference on Declarative Programming*, Marina di Vietri, Italie, pages 37–38. Poligraf Press, Salerno, Italie, 11–14 septembre 1995.

[Cou95d] P. Cousot. Constructing a hierarchy of semantics by abstract interpretation. Dans *Workshop on Logic, Domains and Programming Languages*, Darmstadt, Allemagne, mai 24–27 1995.

[Cou95e] P. Cousot. Set-constraint-based analysis of logic programs by abstract interpretation, conférence invitée. Dans M. Gabbrielli, éditeur, *Proceedings of the Fourth Compulog-network subgroup meeting on Programming Languages*, Marina di Vietri, Italie, pages 1–2, 15 septembre 1995.

[CC95a] P. Cousot et R. Cousot. Compositional and inductive semantic definitions in fixpoint, equational, constraint, closure-condition, rule-based and game-theoretic form, papier invité. Dans P. Wolper, éditeur, *Proceedings of the Seventh International Conference on Computer Aided Verification, CAV '95*, Liège, Belgique, Lecture Notes in Computer Science 939, pages 293–308. Springer-Verlag, Berlin, Allemagne, 3–5 juillet 1995.

[CC95b] P. Cousot et R. Cousot. Formal language, grammar and set-constraint-based program analysis by abstract interpretation. Dans *Proceedings of the Seventh*

ACM Conference on Functional Programming Languages and Computer Architecture, pages 170–181, La Jolla, Californie, 25–28 juin 1995. ACM Press, New York, New York, USA.

[Cou96] P. Cousot. Abstract interpretation. *Symposium on Models of Programming Languages and Computation, ACM Computing Surveys*, 28(2):324–328, 1996.

[Cou97] P. Cousot. Types as abstract interpretations. Dans *Conference Record of the Twentyfourth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages Paris, France, janvier 1992, à paraître. ACM Press, New York, New York, USA.

[CC97] P. Cousot et R. Cousot. Parallel combination of abstract interpretation and model-based automatic analysis of software. À paraître, 1997.

[Cri95] R. Cridlig. Semantic analysis of shared-memory concurrent languages using abstract model-checking. Dans *Proceedings of the ACM Symposium on Partial Evaluation and Semantics-Based Program Manipulation, PEPM '95*, La Jolla, Californie, 21–23 juin 1995. ACM Press, New York, New York, USA, juin 1995.

[Cri96a] R. Cridlig. Implementating a static analyzer of concurrent programs: Problems and perspectives. *LOMAPS Workshop*, Stockholm, Suède, juin 1996. À paraître dans Lecture Notes in Computer Science, 1997.

[Cri96b] R. Cridlig. Semantic analysis of Concurrent ML by abstract model-checking. *INFINITY Workshop*, Pise, Italie, août 1996.

[CG93] R. Cridlig et É. Goubault. Semantics and analysis of Linda-based languages. Dans P. Cousot, M. Falaschi, G. Filé, et A. Rauzy, éditeurs, *Proceedings of the Third International Workshop WSA '93 on Static Analysis*, Padoue, Italie, Lecture Notes in Computer Science 724, pages 72–86. Springer-Verlag, Berlin, Allemagne, 22–24 septembre 1993.

[DOTY96] C. Daws, A. Olivero, S. Tripakis, and S. Yovine. The tool KRONOS. In *Hybrid Systems III, Verification and Control*, pages 208–219. Lecture Notes in Computer Science 1066, Springer-Verlag, 1996.

[DOY94] C. Daws, A. Olivero, and S. Yovine. Verifying ET-LOTOS programs with KRONOS. In D. Hogrefe and S. Leue, editors, *Proceedings 7th. IFIP WG G.1 International Conference of Formal Description Techniques, FORTÉ94*, pages 227–242, Bern, Switzerland, octobre 1994. Formal Description Techniques VII, Chapman & Hall.

[DY95] C. Daws and S. Yovine. Two examples of verification of multirate timed automata with KRONOS. In *Proceedings 1995 IEEE Real-Time Systems Symposium, RTSS'95*, Pisa, Italy, décembre 1995. IEEE Computer Society Press.

[DY96] C. Daws and S. Yovine. Reducing the number of clock variables of timed automata. In *Proceedings 1996 IEEE Real-Time Systems Symposium, RTSS'96*, Washington, DC, USA, décembre 1996. IEEE Computer Society Press.

[DBJ90] A. Debreil, C. Berthet, A. Jerraya, Symbolic computation of VHDL hierarchical descriptions, in Proceedings of *1st European Conference on VHDL Methods*, Marseilles, France, September 1990.

[Deh90] B. Dehbonei. *Génération de code et analyse interprocédurale au sein d'un environnement de programmation parallèle.* Thèse de doctorat, université de Paris 6, décembre 1990.

[Deu91] A. Deutsch. An operational model of strictness properties and its abstraction. In *Proceedings of the 1991 Glasgow University Functional Programming Workshop.* Springer-Verlag, New York, août 1991.

[Deu92a] A. Deutsch. *Modèles opérationnels de langages de programmation et représentations de relations sur des langages rationnels avec application à la détermination statique de propriétés de partages dynamiques de données.* Thèse de doctorat, Université de Paris VI, Paris, 1 avril 1992.

[Deu92b] A. Deutsch. A storeless model of aliasing and its abstraction using finite representations of right-regular equivalence relations. In *Proceedings of the 1992 International Conference on Computer Languages*, pages 2–13, Oakland, California, USA, avril 1992. IEEE Computer Society Press, Los Alamitos, California, USA.

[DMGC92] P. Deverchère, J. C. Madre, J. B. Guignet, M. Currat, Functional abstractiom and formal proof of digital circuits, in Proceedings of *EDAC'92*, Brussels, Belgium, March 1992.

[D90a] B. Dehbonei, C. Laurent, N.Tawbi, R. & S. Kulkarni, *Pmacs : An environement for parallel programming*, In proceedings of the International Symposium, Anaheim, April 1991.

[D90b] B. Dehbonei, *Génération de code et analyse interprocédurale au sein d'un environnement de programmation parallèle*, Thèse de l'univesité Pierre et Marie Curie, Décembre 1990.

[DJ92] B. Dehbonei et P. Jouvelot, Semantical interprocedural analysis by partial symbolic evaluation, à paraître dans les actes de *Workshop on Partial Evaluation and Semantics Based Program Manipulation* (conférence ACM), Juin 1992.

[DTF91] B.Dehbonei, N.Tawbi & P.Feautrier *Parametrized dependence graph : A structure for efficient recompilation inside a parallel programming environment*, IEEE-IMACS PDCom'91, Grèce, 1991.

[EFJ90] P. Ernberg, L. Fredlund, and B. Jonsson. Specification and validation of an overtaking protocol using LOTOS. rapport technique T90006, SICS, Kista, Sweden, 1990.

[Fer88] J.C. Fernandez. *Aldébaran, un système de vérification par réduction de processus communicants*. Thèse, Université Joseph Fourier, Grenoble, mai 1988.

[Fer90] J. C. Fernandez. An implementation of an efficient algorithm for bisimulation equivalence. *Science of Computer Programming*, 13(2-3), mai 1990.

[FQ88] Felipe Fernandez et Patrice Quinton. *Extension of Chernikova's Algorithm For Solving General Mixed Linear Programming Problems* Rapport technique, IRISA, octobre 1988.

[FRV85] J.C. Fernandez, J.L. Richier, et J. Voiron. Verification of protocols specifications using the CESAR system. In *IFIP Workshop on protocols*, juin 1985.

[Fer93] J. Cl. Fernandez. Abstract interpretation and verification of reactive systems. In *Static Analysis*. LNCS 724, Springer Verlag, septembre 1993.

[FKM93] J.-C. Fernandez, A. Kerbrat, and L. Mounier. Symbolic equivalence checking. In *5th Workshop on Computer-Aided Verification (Elounda, Greece)*. LNCS 697, Springer Verlag, juillet 1993.

[FGK+96] Jean-Claude Fernandez, Hubert Garavel, Alain Kerbrat, Radu Mateescu, Laurent Mounier, and Mihaela Sighireanu. Cadp (cæsar/aldebaran development package): A protocol validation and verification toolbox. In Rajeev Alur and Thomas A. Henzinger, editors, *Proceedings of the 8th Conference on Computer-Aided Verification (New Brunswick, New Jersey, USA)*, volume 1102 of *LNCS*, pages 437–440. Springer Verlag, aout 1996.

[FL88] P.G. Frankl et E.J. Weyuker. An applicable family of data flow testing criteria. *IEEE Transactions on Software Engineering*, 14(10), octobre 1988.

[Gar89a] Hubert Garavel. *Compilation et vérification de programmes LOTOS*. Thèse, Université Joseph Fourier, Grenoble, novembre 1989.

[Gar89b] Hubert Garavel. Compilation of Lotos abstract data types. In Son T. Vuong, éditeur, *Proceedings of the 2nd International Conference on Formal Description Techniques FORTE'89 (Vancouver B.C., Canada)*, North Holland, Amsterdam, Décembre 1989.

[GS90] Hubert Garavel et Joseph Sifakis. Compilation and verification of lotos specifications. In L. Logrippo, R. L. Probert, et H. Ural, éditeurs, *Proceedings of the 10th IFIP International Symposium on Protocol Specification, Testing and Verification (Ottawa)*, North Holland, juin 1990.

[Gou93a] É. Goubault. Domains of higher-dimensional automata. Dans E. Best, éditeur, *CONCUR'93, Proceedings of the Fourth International Conference on Concurrency Theory*, Hildesheim, Allemagne, août 1993, Lecture Notes in Computer Science 715, pages 293–307. Springer-Verlag, Berlin, Allemagne, août 1993.

[Gou93b] J. Goubault. Syntax independent connections. In D. *et al.*. Basin, editor, *Workshop on Theorem Proving with Analytic Tableaux and Related Methods*, number MPI-I-93-213. Max Planck Institut für Informatik, 1993.

[Gou93c] J. Goubault. *Démonstration automatique en logique classique : complexité et méthodes*. PhD thesis, École Polytechnique, Palaiseau, France, 1993.

[Gou94a] J. Goubault. A BDD-based skolemization procedure. In M. D'Agostino, editor, *Workshop on Theorem Proving with Analytic Tableaux and Related Methods*, 1994. submitted.

[Gou94b] J. Goubault. Proving with BDDs and Control of Information In A.Bundy, editor, *12th CADE*, 1994. submitted.

[Gou94c] J. Goubault. The complexity of resource-bounded first-order classical logic. In *11th STACS*, 1994.

[Gou95a] É. Goubault. *Geometry of Concurrency*. Thèse de doctorat de l'école polytechnique en informatique, École Polytechnique, Palaiseau, France, 13 novembre 1995.

[Gou95b] É. Goubault. Schedulers as abstract interpretations of higher-dimensional automata. Dans *Proceedings of the ACM Symposium on Partial Evaluation and Semantics-Based Program Manipulation, PEPM'95*, La Jolla, Californie, 21–23 juin 1995, pages 134–145. ACM Press, New York, New York, USA, juin 1995.

[Gou96a] É. Goubault. Duration for truly-concurrent transitions. Dans H. Riis Nielson, éditeur, *Proceedings of the Sixth European Symposium on Programming, ESOP'96*, Linköping, Suède, Lecture Notes in Computer Science 1058, pages 173–187. Springer-Verlag, Berlin, Allemagne, avril 22–26 1996.

[Gou96b] É. Goubault. Higher dimensional automata, Part I – Basic definitions. Rapport de recherche. à paraître, Laboratoire d'Informatique, École Normale Supérieure, Paris, France, 1996.

[GL93] S. Graf and C. Loiseaux. A tool for symbolic program verification and abstraction. In *5th Workshop on Computer-Aided Verification (Elounda, Greece)*. LNCS 697, Springer Verlag, juillet 1993.

[Gra95] S. Graf. Characterization of a sequentially consistent memory and verification of a cache memory by abstraction. *à paraître en Distributed Computing*, 1995.

[Gra89] P. Granger. Static analysis of arithmetical congruences. *International Journal of Computer Mathematics*, 30:165–190, 1989.

[Gra91a] P. Granger. *Analyses sémantiques de congruence.* Thèse de l'École Polytechnique en informatique, LIX, École Polytechnique, Palaiseau, France, 12 juillet 1991.

[Gra91b] P. Granger. Static analysis of linear congruence equalities among variables of a program. In S. Abramsky and T.S.E. Maibaum, editors, *TAPSOFT'91, Proceedings of the International Joint Conference on Theory and Practice of Software Development*, Brighton, UK, Volume 1 (CAAP'91), Lecture Notes in Computer Science 493, pages 169–192. Springer-Verlag, New York, 1991.

[Hal79] N. Halbwachs. *Détermination automatique de relations linéaires vérifiées par les variables d'un programme.* Thèse de 3$^{\text{ème}}$ cycle d'informatique, Université scientifique et médicale de Grenoble, Grenoble, France, 12 mars 1979.

[Hal83] N. Halbwachs. *Analyse statique des propriétés linéaires des systèmes indétermin istes et parallèles.* Rapport Technique 367, IMAG, Grenoble, avril 1983.

[HCRP91] N. Halbwachs, P. Caspi, P. Raymond, et D. Pilaud. Programmation et vérification des systèmes réactifs à l'aide du langage flot de données synchrone LUSTRE. *Technique et Science Informatique*, 10(2), 1991.

[HLca] N. Halbwachs et F. Lagnier. *An experience in proving regular networks of processes by modular model checking.* Rapport Technique SPECTRE L13, IMAG, Grenoble, mars 1990 (to appear in Acta Informatica).

[Hal93] N. Halbwachs. Delay analysis in synchronous programs. In *5th Workshop on Computer-Aided Verification (Elounda, Greece).* LNCS 697, Springer Verlag, juillet 1993.

[Han96] M. Handjieva. STAN: A static analyzer for CLP($\Re$) based on abstract interpretation. Dans R. Cousot et D.A. Schmidt, éditeurs, *Proceedings of the Static Analysis Symposium, SAS'96*, Aix-La-Chapelle, Allemagne, 24–26 septembre 1996, Lecture Notes in Computer Science 1145, pages 383–384. Springer-Verlag, Berlin, Allemagne, 1996.

[HNSY92] T. Henzinger, X. Nicollin, J. Sifakis, et S. Yovine. Symbolic model-checking for real-time systems. In *LICS'92*, juin 1992.

[HPOG89] N. Halbwachs, D. Pilaud, F. Ouabdesselam, et A.C. Glory. Specifying, programming and verifying real-time systems, using a synchronous declarative language. In *Workshop on automatic verification methods for finite state systems, LNCS 407*, Springer Verlag, juin 1989.

[HRR91] N. Halbwachs, P. Raymond, et C. Ratel. Generating efficient code from dataflow programs. In *Third International Symposium on Programming Language Implementation and Logic Programming,* Passau, aout 1991.

[HLR92] N. Halbwachs, F. Lagnier, and C. Ratel. Programming and verifying real-time systems by means of the synchronous data-flow programming language LUSTRE. *IEEE Transactions on Software Engineering, Special Issue on the Specification and Analysis of Real-Time Systems*, septembre 1992.

[HPR94] N. Halbwachs, Y.-E. Proy, and P. Raymond. Verification of linear hybrid systems by means of convex approximations. In B. LeCharlier, editor, *International Symposium on Static Analysis, SAS'94*, Namur (Belgium), septembre 1994. LNCS 864, Springer Verlag.

[HMP95] N. Halbwachs, F. Maraninchi, and Y. E. Proy. The railroad crossing problem, modeling with Hybrid Argos - Analysis with Polka. In *Second European Workshop on Real-Time and Hybrid Systems*, Grenoble (France), juin 1995.

[HR96] N. Halbwachs and P. Raymond. On the use of approximation in symbolic model-checking. Technical report, Verimag Laboratory, janvier 1996.

[HNSY92] T. Henzinger, X. Nicollin, J. Sifakis, and S. Yovine. Symbolic model-checking for real-time systems. In *LICS'92*, juin 1992.

[HO82] C.M. Hoffmann et M.J. O'Donnell. Pattern matching in trees. Journal of the ACM, 29(1), pp.68–95, January 1982.

[HD93] A. J. Hu and D. L. Dill. Efficient verification with BDDs using implicitly conjuncted invariants. In *Fifth Conference on Computer-Aided Verification, CAV'93*, Elounda (Greece), juillet 1993. LNCS 697, Springer Verlag.

[Jon 86] C. B. Jones *Systematic Software Development using VDM.* Prentice Hall 1986.

[JD89a] P.Jouvelot, B.Dehbonei *A Unified Semantic Approach for the Vectorization and Parallelization of Generalized Reductions* ACM Sigarch ICS'89, Crete Juin 1989

[JD89b] P.Jouvelot, B.Dehbonei *Recursive Pattern Matching on Concrete Data Types* ACM Sigplan Notices, novembre 1989

[HS94] M.J. Harrold et M.L. Soffa. Efficient computation of interprocedural definition-use chains. In *ACM Transactions on Programming Languages and Systems*, 16(2):175–204, mars 1994.

[Ka76] M. Karr. Affine relationships among variables of a program. *Acta Informatica*, 6:133–151, 1976.

[Koz83] D. Kozen. Results on the propositional μ-calculus. In *Theoretical Computer Science*. North-Holland, 1983.

[Lan92] W.A. Landi. Interprocedural aliasing in presence of pointers. PhD Thesis, Rutgers University, janvier 1992.

[L90] C. Laurent, *Vectorisation automatique de boucles comportant des instructions de branchement*, Thèse de l'univesité Pierre et Marie Curie, Septembre 1990.

[Lev92] H. Le Verge. A note on Chernikova's Algorithm. Rapport de Recherche, IRISA/INRIA, 1992.

[LCM92] B. Lin, O. Coudert, J. C. Madre, Symbolic prime generation for multiple-valued functions, in Proceedings of *29th DAC*, Anaheim CA, USA, June 1992.

[LGS+95] C. Loiseaux, S. Graf, J. Sifakis, A. Bouajjani, and S. Bensalem. Property preserving abstractions for the verification of concurrent systems. *Formal Methods in System Design, Vol 6, Iss 1*, janvier 1995.

[Loi94] C. Loiseaux. *Vérification symbolique de programmes réactifs à l'aide d'abstractions.* PhD thesis, Université Joseph Fourier, Grenoble, février1994.

[MB88] J. C. Madre, J. P. Billon, Proving circuit correctness using formal comparison between expected and extracted behaviour, in Proceedings of *25th DAC*, Anaheim CA, USA, July 1988.

[MC89a] J. C. Madre, O. Coudert, Formal verification of digital circuits using a propositional theorem prover, in Proceedings of *IFIP Working Conference on the CAD Systems Using AI Techniques*, Tokyo, June 1989.

[Mad89] J. C. Madre, Benchmarks for tautology checking - experimental results, in *Formal VLSI Correctness Verification*, L. J. M. Claesen Editor, North-Holland, pp. 119–123, November 1989.

[MC89b] J. C. Madre, O. Coudert, Automating the diagnosis and the rectification of design errors with PRIAM, in Proceedings of *ICCAD'89*, Santa Clara CA, USA, November 1989.

[MCM+90] J. C. Madre, O. Coudert, M. Currat, A. Debreil, C. Berthet, The formal verification chain at BULL, in Proceedings of *EUROASIC Conference*, Paris, France, June 1990.

[Mad90] J. C. Madre, *PRIAM, un outil de preuve formelle de circuits digitaux*, Thèse de troisième cycle (PhD. Thesis), Ecole Nationale Supérieure des Télécommunications, June 1990.

[MC90] J. C. Madre, O. Coudert, A complete reasoning maintenance system based on typed decision graphs, *BULL Research Report* N°90004, 1990.

[MABC91] J. C. Madre, F. Anceau, J. P. Billon, M. Currat, Formal verification in an industrial design environment, in *The International Journal on Computer Aided VLSI Design: Special Issue on Formal Methods in VLSI Design*, P. Prinetto Editor, 1991.

[MC91] J. C. Madre, O. Coudert, A logically complete reasoning maintenance system based on a logical constraint solver, in Proceedings of *IJCAI'91*, Sydney, Australia, August 1991.

[Mas91] F. Masdupuy. Using abstract interpretation to detect array data dependencies. In *Proceedings of the International Symposium on Supercomputing*, pages 19–27, Fukuoka (Japan), novembre 1991.

[Mas93a] F. Masdupuy. *Analyse sémantique relationnelle des indices de tableaux par congruence et trapézoïdes rationnels.* Thèse de l'École Polytechnique en informatique, LIX, École Polytechnique, Palaiseau, France, 21 décembre 1993.

[Mas93b] F. Masdupuy. Semantic analysis of interval congruences. Dans D. Bjø rner, M. Broy, et I.V. Pottosin, éditeurs, *Proceedings of the International Conference on Formal Methods in Programming and their Applications*, Academgorodok, Novosibirsk, Russie, Lecture Notes in Computer Science 735, pages 142–155. Springer-Verlag, Berlin, Allemagne, juin 28–juillet 2, 1993.

[Mau94] L. Mauborgne. Abstract interpretation using TDGs. Dans B. Le Charlier, éditeur, *Proceedings of the Static Analysis Symposium, SAS'94*, Namur Belgique, 20–22 septembre 1994, Lecture Notes in Computer Science 864, pages 363–379. Springer-Verlag, Berlin, Allemagne, 1994.

[Mer90] N. Mercouroff. *Analyses sémantiques de processus communiquants.* Thèse de l'école polytechnique, September 1990.

[Mon91] B. Monsuez. An attempt to find polymorphic types by abstract interpretation. *BIGRE, Actes JTASPEFL'91, Bordeaux*, 74:18–26, octobre 1991.

[Mon92a] B. Monsuez. Fractional types. *BIGRE, Analyse Statique WSA '92, Bordeaux,* IRISA, Rennes, France, 81–82:274–284, 23–25 septembre 1992.

[Mon92b] B. Monsuez. Polymorphic typing by abstract interpretation. Dans *Proceedings of the Twelfth Foundations of Software Technology and Theoretical Computer Science Conference*, pages 127–138, New Delhi, Inde, 18–20 décembre 1992, Lecture Notes in Computer Science 652, 1992. Springer-Verlag, Berlin, Allemagne.

[Mon93a] B. Monsuez. Polymorphic types and widening operators. Dans P. Cousot, M. Falaschi, G. Filé, et A. Rauzy, éditeurs, *Proceedings of the Third International Workshop WSA '93 on Static Analysis*, Padoue, Italie, Lecture Notes in Computer Science 724, pages 267–281. Springer-Verlag, Berlin, Allemagne, 22–24 septembre 1993.

[Mon93b] B. Monsuez. Polymorphic typing for call-by-name semantics. Dans D. Bjørner, M. Broy, et I.V. Pottosin, éditeurs, *Proceedings of the International Conference on Formal Methods in Programming and their Applications*, Academgorodok, Novosibirsk, Russie, Lecture Notes in Computer Science 735, pages 156–169. Springer-Verlag, Berlin, Allemagne, juin 28–juillet 2, 1993.

[Mon94] B. Monsuez. *Typage par interprétation abstraite.* Thèse de doctorat de l'école polytechnique en informatique, École Polytechnique, Palaiseau, France, 13 février 1994.

[Mon95a] B. Monsuez. System F and abstract interpretation. Dans A. Mycroft, éditeur, *Proceedings of the Static Analysis Symposium, SAS '95*, Glasgow, Royaume Uni, 25–27 septembre 1995, Lecture Notes in Computer Science 983, pages 279–295. Springer-Verlag, Berlin, Allemagne, 1995.

[Mon95b] B. Monsuez. Using Abstract interpretation to define a strictness type inference system. Dans *Proceedings of the ACM Symposium on Partial Evaluation and Semantics-Based Program Manipulation, PEPM '95*, La Jolla, Californie, 21–23 juin 1995, pages 122–133. ACM Press, New York, New York, USA, juin 1995.

[NSY92] X. Nicollin, J. Sifakis, and S. Yovine. Compiling real-time specifications into extended automata. *IEEE TSE Special Issue on Real-Time Systems*, 18(9):794–804, septembre 1992.

[QS82] J. P. Queille et J. Sifakis. Specification and verification of concurrent systems in CESAR. In *International Symposium on Programming, LNCS 137*, Springer Verlag, avril 1982.

[RG 92] RAISE Group, *The RAISE Specification Language*, Prentice Hall, 1992.

[RR92] R. Ramesh et I.V. Ramakrishnan. Nonlinear pattern matching in trees. Journal of the ACM, 39(2), pp.295–216, April 1992.

[Ras90] A. Rasse. *CLEO : diagnostic des erreurs en* XESAR. These, Institut National Polytechnique, Grenoble, juin 1990.

[RHR91] C. Ratel, N. Halbwachs, et P. Raymond. Programming and verifying critical systems by means of the synchronous data-flow programming language LUSTRE. In *ACM-SIGSOFT'91 Conference on Software for Critical Systems,* New Orleans, décembre 1991.

[Rat92] C. Ratel. Définition et réalisation d'un outil de vérification formelle de programmes Lustre: Le système Lesar. Thèse, Université Joseph Fourier, Grenoble, juin 1992.

[Rod88] C. Rodriguez. *Spécification et validation de systèmes en* XESAR. Thèse, Institut National Polytechnique, Grenoble, mai 1988.

[RRSV87a] J. L. Richier, C. Rodriguez, J. Sifakis, et J. Voiron. Verification in XESAR of the sliding window protocol. In *IFIP WG-6.1 7th. International Conference on Protocol Specification, Testing and Verification*, North Holland, 1987.

[RRSV87b] J.L. Richier, C Rodriguez, J. Sifakis, et J. Voiron. XESAR *User's Guide*. Rapport Technique Spectre C6, LGI/IMAG, Grenoble, septembre 1987.

[Spivey 89] J. M. Spivey, *The Z Notation*, (Prentice Hall, 1989)

[Str88] J. Stransky. *Analyse sémantique de structures de données dynamiques avec application au cas particulier de langages LISPiens.* Thèse de doctorat en science, Université de Paris-sud, Orsay, 28 juin 1988.

[Suf85] B. Suffrin. *Z Handbook, draft 1.1.* Oxford Programming Research Group, octobre 1985.

[T91] N. Tawbi, *Parallélisation automatique : estimation des durées d'exécution et allocation statique de processeurs*, Thèse de l'univesité Pierre et Marie Curie, Avril 1991.

[TF92] N. Tawbi, P. Feautrier, Processor allocation and loop scheduling on multiprocessor computers, actes de ICS'92 ACM-Conference, Juillet 1992.

[TSL+90] H. Touati, H. Savoj, B. Lin, R. K. Brayton, and A. Sangiovanni-Vincentelli. Implicit state enumeration of finite state machines using BDDs. In *International Conference on Computer Aided Design (ICCAD)*, novembre 1990.

[TY96] S. Tripakis and S. Yovine. Analysis of timed systems based on time–abstracting bisimulations. In *Proceedings 8th Conference Computer-Aided Verification, CAV'96*, pages 232–243, Rutgers, NJ, juillet 1996. Lecture Notes in Computer Science 1102, Springer-Verlag.

[Ved95] F. Védrine. Binding-time analysis and strictness analysis by abstract interpretation. Dans A. Mycroft, éditeur, *Proceedings of the Static Analysis Symposium, SAS '95*, Glasgow Royaume Uni, 25–27 septembre 1995, Lecture Notes in Computer Science 983, pages 400–417. Springer-Verlag, Berlin, Allemagne, 1995.

[Ver86] D. Vergamini. *Verification by means of observational equivalence on automata.* Rapport de Recherche 501, INRIA, 1986.

[WM94] R. Wilhelm et D. Maurer. *Les compilateurs — théorie, construction, génération.* Masson, Paris, 1994.

[Yov93] S. Yovine. Méthodes et outils pour la vérification symbolique de systèmes temporisés. Thèse, Institut National Polytechnique de Grenoble, mai 1993.

Sigles utilisés:

- CADE: International Conference on Automated Deduction
- DAC: ACM/IEEE Design Automation Conference
- ICCAD: ACM/IEEE International Conference on Computer Aided Design
- ICCD: IEEE International Conference on Computer Design
- EDAC: IEEE European Design Automation Conference

- CAV: International Workshop on Computer-Aided Verification
- IJCAI: International Join Conference on Artificial Intelligence
- TSI: Techniques et Sciences de l'Informatiques