Analyse statique par interprétation abstraite de systèmes hybrides.

Olivier Bouissou

Thèse présentée pour obtenir le grade de

DOCTEUR DE L'ÉCOLE POLYTECHNIQUE

Spécialité : Informatique

par

Olivier Bouissou

Analyse statique par interprétation abstraite de systèmes hybrides

Soutenue le 23 septembre 2008 devant le jury composé de :

Patrick Cousot	École Normale Supérieure, Paris	*Rapporteur*
Gilles Villard	École Normale Supérieure, Lyon	*Rapporteur*
Michel Kieffer	École Supérieure d'Électricité, Gif-sur-Yvette	*Président du Jury*
Sriram Sankaranarayanan	NEC Laboratories America, Princeton	*Examinateur*
Eric Goubault	École Polytechnique, Palaiseau	*Examinateur*
Matthieu Martel	Université de Perpignan Via Domitia, Perpignan	*Directeur de thèse*

Remerciements

Ce travail de thèse n'aurait pas été possible, ou aurait été très différent, sans la présence de nombreuses personnes autour de moi que je souhaite ici remercier du fond du coeur.

En tout premier lieu, merci mille fois à mon directeur de thèse, Matthieu Martel, de m'avoir mis sur de si bons rails en me donnant ce sujet d'étude. Il m'a toujours soutenu et poussé pour aller plus loin, et m'a beaucoup transmis au cours de ces trois années. Merci Matthieu de t'être autant investi dans mon travail, je sais que tout ça n'aurait pas été possible sans toi. Je remercie également le directeur du laboratoire MeASI, Eric Goubault, qui m'a permis de travailler dans les meilleures conditions dont on puisse rêver en tant que doctorant. Merci également Eric pour tes conseils toujours avisés et les nombreux chocolats que tu apportes régulièrement.

Je remercie chaleureusement les rapporteurs de cette thèse, Mrs Cousot et Villard, qui ont accepté de relire en détail le manuscrit et dont les commentaires ont été plus qu'utiles. J'ai été très honoré d'avoir de tels rapporteurs et je suis flatté que vous ayez apprécié mes travaux. Un grand merci également à Michel Kieffer d'avoir accepté de faire partie de mon jury de thèse et dont les discussions au début de ma thèse ont grandement contribué à l'amélioration de GRKLib, ainsi qu'à Sriram Sankaranarayanan qui a pris sur son temps de vacances pour venir à ma soutenance.

Je n'aurai jamais pu faire cette thèse sans l'accueil formidable qui m'a été réservé au sein des laboratoires LSL puis MeASI, tant d'un point de vue scientifique qu'humain. Ces trois années auraient été beaucoup plus dures sans Alex, Nico et Michel, qui sont, je l'espère, devenus bien plus que des collègues, surtout grâce aux parties de belote endiablées qui agitaient certaines de nos pauses déjeuner. Vous avez joué un rôle plus qu'important dans la réussite de cette aventure, merci pour tout. Je n'oublie bien sûr pas les autres, Emmanuel qui n'a jamais séché malgré mes nombreuses questions pointues en maths, Sarah, Muriel et Patricia qui ont apporté une touche féminine bienvenue dans ce laboratoire d'informaticiens et tous ceux qui font de ce laboratoire un lieu de travail très sympa.

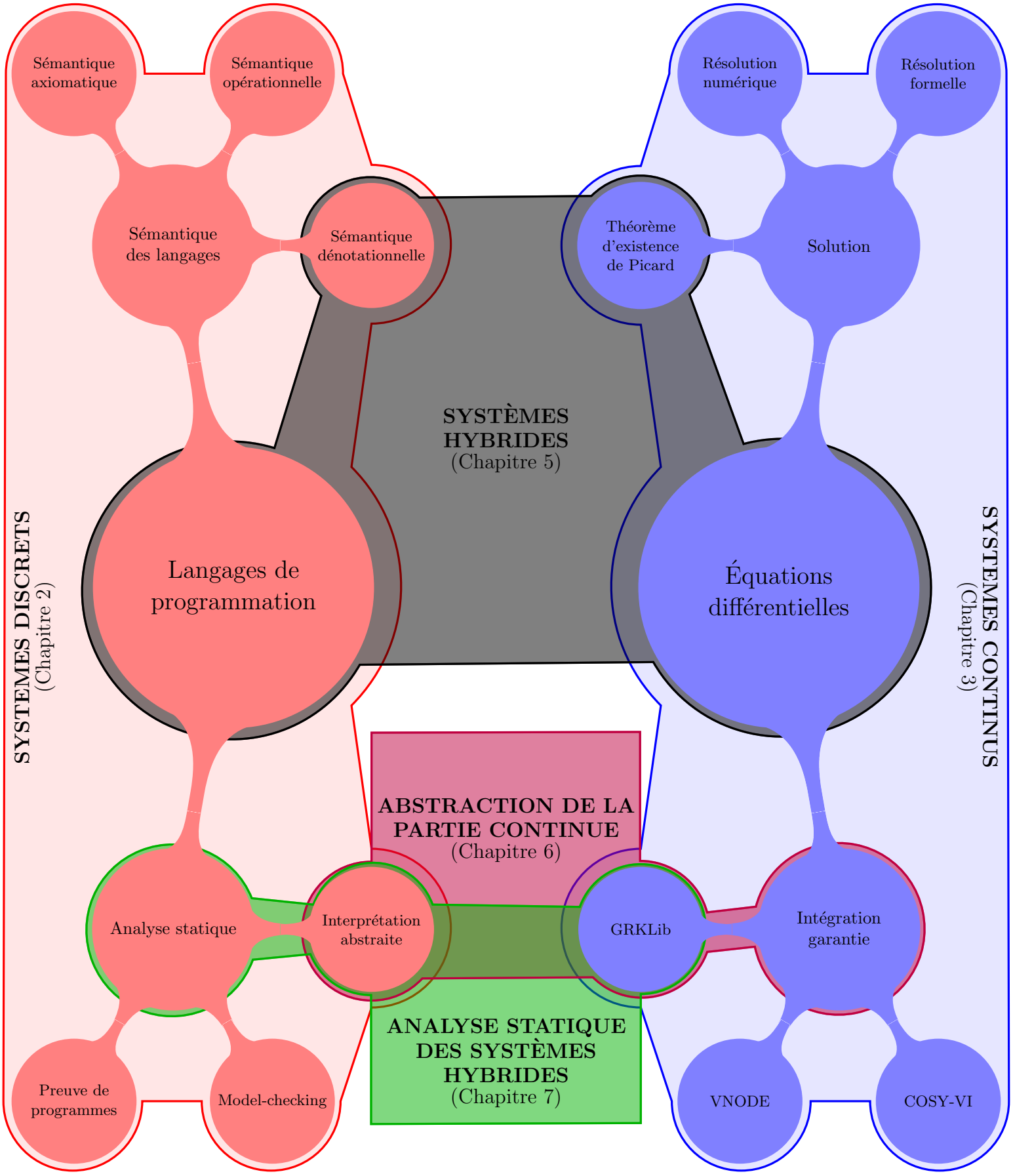
Un grand merci enfin pour mes parents, grands-parents, soeurs, amis, qui m'ont toujours soutenus tant financièrement que moralement au cours de mes (trop ?) longues années d'étude. Même si je n'ai jamais réussi à vous expliquer ce que je fais, vous avez fait semblant de comprendre ce qui est malgré tout très rassurant.

Enfin, je ne pourrai jamais assez te remercier, Camille, d'avoir été là à côté de moi pendant ces 3 ans, de m'avoir soutenu lors des périodes difficiles et de ne pas avoir trop râlé quand je rentrais tard à cause des deadlines d'article approchantes. Tout ce travail t'est dédié.

Table des matières

Un bon croquis vaut mieux qu'un long discours.

Napoleon Bonaparte



CHAPITRE 1

Introduction

1.1 Bref rappel historique

Le dictionnaire Robert donne la définition ci-dessous pour un système critique :

> **Critique,** adj. (informatique).
> 1. Qui implique des suites de grande importance, dans un sens favorable ou défavorable.
> 2. Qui comporte un danger, des risques.

On comprend dès lors que, dans les domaines où les risques sont très importants (d'un point de vue humain ou financier), l'industrie ait pendant longtemps hésité à confier les tâches les plus critiques à un ordinateur. Ce n'est que dans les années 80 que l'industrie aéronautique a décidé de confier à un calculateur le programme de vol des avions. À l'époque, le peu de confiance que l'on avait dans le comportement d'un programme informatique faisait craindre le pire pour ces nouvelles générations d'avions (on parlait même de "Désastre assisté par ordinateur") [Mel94]. Depuis, le nombre d'accidents aéronautiques graves n'a cessé de diminuer et la confiance en l'informatique embarquée s'est accrue au point qu'on a lui quasiment confié toutes les tâches critiques. Cependant, cette assistance (pour ne pas dire prise de contrôle) informatique apporte de nouveaux dangers qu'il est primordial de bien comprendre pour pouvoir les maîtriser. Dans cette section, nous proposons un rapide tour d'horizon de ces nouveaux dangers en utilisant trois exemples célèbres de bugs informatiques.

1.1.1 Échec du missile Patriot

Au cours de la première guerre du Golfe, les militaires américains utilisent de nombreux missiles Patriot pour protéger les points stratégiques des missiles Scud irakiens. Le missile Patriot est un missile sol-air complètement automatique qui cherche et intercepte les engins ennemis dans une certaine zone de l'espace. Dans la nuit du 25 février 1991, un des missiles Patriot lancé depuis Dhahran (Arabie Saoudite) a échoué dans l'interception d'un missile Scud, provoquant la mort de 28 soldats américains. L'enquête officielle menée par l'armée américaine [GAO92] est arrivée aux conclusions suivantes.

Pour intercepter une missile Scud, la batterie de tir des Patriot fonctionne comme suit. Le radar repère dans une large zone de l'espace les objets volants. En utilisant une base de données des propriétés des missiles Scud, l'ordinateur de bord détecte si un des objets est potentiellement un missile irakien. Si tel est le cas, il calcule dans quelle zone de l'espace le missile devrait se

trouver ensuite et se concentre alors sur cette zone : si le radar y retrouve un objet avec les bonnes propriétés, alors il conclut qu'il s'agissait bien d'un missile Scud et il utilise le même algorithme pour prédire sa trajectoire et l'intercepter. Pour prédire la trajectoire du missile, le système utilise deux valeurs : la vitesse du missile mesurée par le radar et le temps de la détection par le radar. La vitesse est exprimée par un nombre flottant codé sur 24 bits (la précision maximale de l'ordinateur de contrôle des Patriot). Le temps est encodé par un entier exprimant le nombre de dixièmes de secondes écoulées depuis le dernier redémarrage du système. Cet entier est ensuite convertit en une valeur flottante sur 24 bits pour calculer la trajectoire du missile. Lors d'une utilisation très prolongée du système Patriot, l'entier représentant le temps devient très important et la conversion en une valeur flottante cause une perte de précision qui influe directement sur la précision de l'estimation de trajectoire. Ainsi, après une utilisation de la batterie durant 20 heures, l'erreur sur la mesure du temps est de 0.0687 secondes (soit moins de $10^{-4}\%$) ce qui cause une erreur de 137 mètres sur la prédiction de la trajectoire du Scud. Le 25 février 1991, la batterie de missiles Patriot fonctionnait depuis 100 heures, l'erreur dans la prédiction de la trajectoire était de 687 mètres et le missile n'a pas été intercepté. La perte de précision lors de la conversion d'un entier vers un nombre flottant codé sur 24 bits est donc la seule cause de l'échec du missile Patriot.

1.1.2 Explosion d'Ariane 5

Le 04 juin 1996, le nouveau lanceur Ariane 5 est utilisé pour la première fois. Le lancement se déroule comme prévu pendant 36 secondes, la fusée s'élevant selon son itinéraire de vol. Peu après la 37ème seconde, le lanceur dévie soudainement de son chemin, puis explose, causant l'échec de la mission et une perte sèche de 700 millions d'euros pour Ariane Espace. Le groupe d'enquête indépendant [Boa96] mis en place après cet incident est arrivé aux conclusions suivantes.

La fusée Ariane 5 contient deux systèmes de référence inertiel (SRI) : un actif et un de secours. Chaque SRI possède un ordinateur qui calcule la vitesse et l'angle d'inclinaison à partir de valeurs transmises par des capteurs gyroscopiques et des accéléromètres. Les valeurs calculées par le SRI sont transmises à l'ordinateur central qui commande le vol, et notamment l'inclinaison de la fusée via des actionneurs hydrauliques. Les programmes embarqués dans les deux SRI de la fusée Ariane 5 sont les mêmes que ceux embarqués dans Ariane 4. Lors du premier vol d'Ariane 5, les deux SRI se sont déclarés en échec à cause d'une exception déclenchée lors de la conversion d'un nombre à virgule flottante en un entier non signé. Rappelons qu'un nombre à virgule flottante (ou nombre flottant) est utilisé pour représenter, sur un nombre fini de bits, un nombre réel. La norme IEEE754 [P7585] définit le format de représentation de ces nombres comme une mantisse m suivi d'un exposant e de telle sorte que le nombre correspondant est $m \times 2^e$, le nombre de bits utilisés pour la mantisse et l'exposant étant variable en fonction de l'architecture utilisée et de la précision voulue. Dans le cas d'Ariane 5, la conversion n'était pas protégée et le nombre flottant était supérieur au plus grand entier représentable en machine. Elle a donc produit un nombre non significatif qui a été interprété par l'ordinateur de vol comme l'altitude de la fusée. L'ordinateur a donc réagi en ordonnant aux moteurs de redresser la fusée, ce qui a causé sa déviation et ensuite son explosion. Le plus intéressant dans ce bug est que l'erreur de conversion a eu lieu car les valeurs fournies par les capteurs étaient plus élevées que prévues. Le nombre de bits nécessaires pour la représentation des nombres entiers étaient en effet calibré pour la fusée Ariane 4 et les valeurs mesurées par les capteurs d'Ariane 5 étaient plus élevées car la vitesse horizontale était supérieure.

Remarquons que dans ce cas le bug a été détecté a posteriori par une analyse manuelle du code ainsi que par une analyse automatique via le logiciel ASTREE développé à l'École Normale Supérieure.

1.1.3 Blocage du USS Yorktown

Le navire USS Yorktown est un croiseur lance-missiles de l'armée américaine, lancé en 1984. Il est équipé d'un système de contrôle et maintenance automatique qui gère notamment la propulsion des moteurs. En Septembre 1997, le système informatique s'est bloqué entraînant l'immobilisation

du navire pendant plusieurs heures. L'enquête menée a posteriori a révélé que la cause de l'incident était une mauvaise utilisation d'un programme de gestion à distance de base de données. En effet, l'opérateur chargé de rentrer les données collectées dans la base a accidentellement tapé le chiffre 0. Cela a entraîné une division par 0 dans le programme, et en conséquence un crash du système d'opération sous-jacent. Les programmes de gestion de la propulsion se sont donc trouvés bloqués jusqu'à la réparation du système.

La division par 0 est un exemple classique d'erreur à l'exécution (*Running Time Error*, ou RTE) qui bloque complètement le programme en levant une exception. Dans le cas des deux bugs précédents, l'erreur n'avait pas entraîné de blocage du système mais seulement une mauvaise exécution due à une imprécision dans les calculs. Le cas des RTE bloquantes comme la division par 0 ou les erreurs de segmentation (qui apparaissent quand on essaie d'accéder à une zone mémoire indisponible) est donc très problématique et prouver leur absence est un enjeu capital, d'autant plus que c'est souvent en exploitant ce genre de bugs que des failles de sécurité sont trouvées.

1.1.4 Point commun entre ces trois bugs

Dans tous ces exemples, c'est un bug dans le programme embarqué qui a eu des conséquences très importantes sur le système dans son ensemble. Cependant, si ces conséquences ont été si grandes, c'est précisément parce que ces programmes sont des programmes embarqués qui interagissent avec un environnement extérieur, et dans chaque cas, c'est ce lien qui a été défaillant.

Dans le cas du missile Patriot, l'erreur de calcul du programme est très localisée, elle est pourtant la cause d'une grande perte humaine. En effet, les décisions prises par le lanceur du missile Patriot étaient fondées sur les résultats du programme qui contenait donc des erreurs. Ces erreurs ont donc amené le lanceur à prendre une mauvaise décision, c'est-à-dire qu'il a mal contrôlé les actionneurs permettant de viser et lancer le missile qui a donc manqué sa cible. De même, dans le cas de l'USS Yorktown, la division par 0 était très localisée. Cependant, en entraînant le crash du système sous-jacent, elle a empêché le programme de navigation de fonctionner et a donc fait s'immobiliser le navire. Encore une fois, les actionneurs qui dirigent la propulsion n'ont pas pu être activés à cause d'une erreur a priori bénigne dans un programme embarqué. Le cas de l'Ariane 5 diffère légèrement, mais révèle une autre caractéristique des programmes embarqués. Le bug vient en effet d'une mauvaise adéquation entre un capteur et le logiciel embarqué qui récupère l'information fournie par ce capteur. Le logiciel attendait des valeurs dans un plage de valeurs définie statiquement, alors que ce capteur renvoyait des valeurs bien plus grandes, tout simplement parce que la fusée allait plus vite qu'initialement prévu par le programme. Cette incompréhension entre le programme et le monde physique extérieur est la seule cause de l'explosion d'Ariane 5.

On voit donc que dans tous ces cas, une erreur du programme a eu une conséquence très forte (financière pour Ariane 5 et le USS Yorktown, humaine pour le missile Patriot) à cause de l'interaction entre ce programme et les capteurs et/ou actionneurs qui lui permettent de communiquer avec le monde extérieur. Il est donc primordial, avant d'utiliser des programmes dans des systèmes embarqués, de s'assurer de leur bon fonctionnement.

1.2 Conception et vérification des programmes embarqués critiques

Pour mener à bien la vérification des programmes embarqués, il est nécessaire de bien comprendre leur mode de conception. Dans cette section, nous nous intéressons au cycle de développement dit "en V" des programmes critiques ainsi qu'aux techniques mises en place au cours de ce développement pour s'assurer de leur bon fonctionnement.

1.2.1 Cycle de développement des logiciels embarqués

Le modèle le plus classique (mais également le plus révélateur) de développement d'un logiciel est le cycle en V [FM92, Mei98, Sto96]. Il est présenté graphiquement sur la figure 1.1. Le cycle

en V comporte deux phases, chacune décomposée en plusieurs tâches qui se font face.

La phase de conception compose la partie gauche du V, et doit être lue de haut en bas (voir figure 1.1). Elle représente la définition, décomposition et conception du système au cours de trois tâches successives. La première est la tâche de spécification du système global, soit dans un langage de haut niveau (Matlab/Simulink ou Lustre/SCADE sont les langages les plus utilisées) soit par une spécification moins formelle avec des langages de description comme UML. Cette spécification sert à étudier le système dans son ensemble à des fins de simulation, et elle comprend généralement une description fonctionnelle du programme ainsi que de l'environnement physique dans lequel le programme doit être exécuté. La seconde tâche est la tâche de conception où le système est découpé en sous-systèmes indépendants dont les interactions sont spécifiées. A ce stade on utilise encore un langage de spécification de haut niveau et les tests d'intégration sont mis en place. Enfin, la tâche de développement consiste en le codage des différents composants précédemment définis dans un langage de bas niveau (les langages les plus utilisés sont le C ou ADA). Chaque composant est codé indépendamment des autres, souvent par des équipes différentes ne communiquant pas entre elles. On peut noter qu'à ce niveau, les spécifications des sous-systèmes physiques ne sont pas prises en compte et que seules les parties discrètes du système global sont encodées.

La phase d'intégration et vérification compose la partie droite du V, et doit être lue de bas en haut (voir la figure 1.1). Elle représente la reconstruction du système global à partir des composants précédemment développés ainsi que sa validation en regard des spécifications énoncées durant la phase de spécification. La phase d'intégration est elle aussi divisée en trois tâches. La première intervient au moment du développement des composants et consiste en la vérification de chaque composant individuellement. Cette vérification est effectuée sur des programmes écrits dans un langage de bas niveau et les techniques de test unitaire permettent de valider le comportement de chaque fonction. Les méthodes formelles d'analyse statique sont parallèlement de plus en plus utilisées pour cela. Ensuite, les composants sont assemblés les uns aux autres durant la tâche de construction du système, et des tests d'intégration sont menés. Il s'agit notamment de tester les communications entre les composants du système. Enfin, la tâche de validation du système gère les tests fonctionnels pour valider le système vis-à-vis des exigences de haut niveau fournies soit par le client soit par des contraintes de performance. À ce stade, il s'agit essentiellement de mener des tests sur le système dans son ensemble et les méthodes formelles ne sont que rarement utilisées.

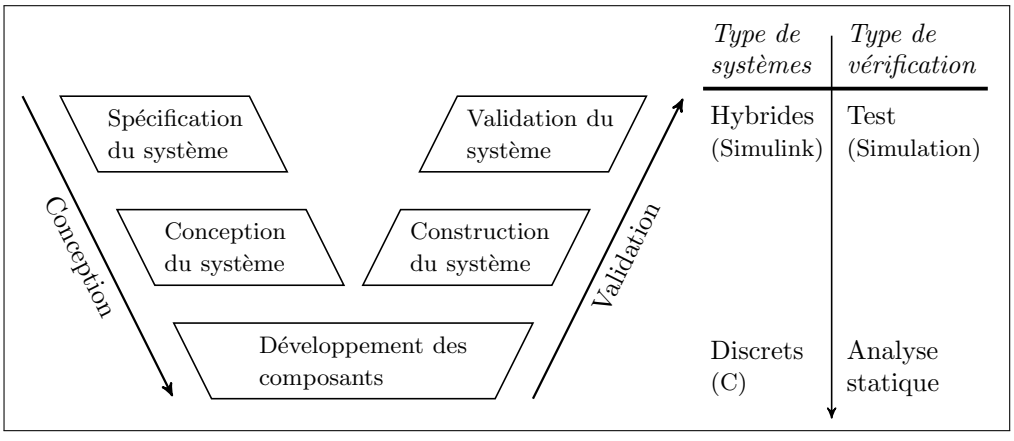


FIG. 1.1 – Cycle de développement en V.

On voit donc qu'aux différents stades du développement du logiciel des techniques très différentes sont utilisées pour s'assurer de son bon fonctionnement : le test pour les phases de spécification

et de conception, l'analyse statique (en plus du test) pour la phase de développement. Examinons rapidement les différences entre ces deux approches.

1.2.2 Test : validation dynamique

Le test apparaît partout dans le cycle de développement des logiciels critiques embarqués. Au niveau de la spécification, des tests de validation sont mis en place pour vérifier, par exemple, le comportement du système en réponse à une variation de l'environnement ("est-ce que la voiture freine lorsque le conducteur appuie sur la pédale de frein ?"). Ce genre de test est souvent très coûteux car il est joué sur l'ensemble du système dans des conditions le plus proche possible des conditions d'utilisation réelles. Par exemple, pour tester le fonctionnement de la phase d'approche du Véhicule Automatique de Transfert européen (en anglais *Automated Transfer Vehicle* ou ATV) vers la station spatiale internationale, EADS/ST a utilisé une piscine de 600 mètres de long pour simuler, avec un prototype de l'ATV, les conditions d'apesanteur. Clairement, des méthodes permettant de s'affranchir de certains de ces tests en apportant des preuves de bon fonctionnement seraient très utiles.

Au niveau le plus bas du cycle en V, des tests unitaires sont menés pour valider chaque composant individuellement. On distingue généralement trois techniques pour le test unitaire. Le test aléatoire consiste à donner, au hasard, des valeurs aux entrées du programme (on parle de *jeu de test*) ; en exécutant de nombreux tirages, on peut ainsi atteindre une confiance relativement élevée dans la fonction. Cependant, si certaines zones sensibles ne sont atteintes qu'avec une probabilité faible par rapport aux valeurs d'entrée, ce genre de test aura du mal à les détecter. Le test fonctionnel génère les jeux de test en fonction des spécifications de la fonction sous test ; des valeurs seront choisies pour chaque région du domaine d'entrée où le programme doit avoir un comportement particulier. Enfin, le test structurel permet de déterminer les valeurs d'entrée par une analyse du code source de la fonction ; on s'assure alors que tous les différents comportements possibles (ou au moins un pourcentage élevé d'entre eux) ont été exécutés. Dans la suite, nous détaillerons les techniques de tests structurels uniquement, car ce sont les plus étudiées et les plus utilisées en pratique pour les logiciels critiques.

But du test structurel. Le but d'un jeu de test est de fournir des données d'entrée pour la fonction qui permettent de vérifier chaque comportement réel de cette fonction. Pour le test structurel, la notion de comportement est liée au code source de la fonction sous test. On distingue là encore deux types de test : le test orienté flot de contrôle et le test orienté flot de données. Le test orienté flot de données cherche à couvrir toutes les relations entre la définition d'une variable et son utilisation. Au contraire, le test orienté flot de contrôle s'intéresse plus à la structure du programme : on dispose alors de plusieurs critères de couverture. Le critère *instructions* cherche à faire exécuter toutes les instructions simples du programme, le critère *tous-les-branchements* cherche à exécuter chaque instruction conditionnelle du programme avec une condition vraie et une condition fausse, et le critère *tous-les-chemins* cherche à exécuter tous les chemins du graphe de flot de contrôle du programme. La figure 1.2 représente les exécutions d'un programme dont les valeurs d'entrées sont comprises entre 0 et 10 pour un jeu de test de 4 valeurs. Ce jeu de test semble montrer que le programme n'entre jamais dans les zone interdites, présentées en rouge sur la figure. Les points sont les valeurs prises par la variable x aux différentes lignes du programme. En pratique, le critère *tous-les-chemins* reste le meilleur critère pour trouver des bugs, mais il est coûteux car le calcul des données nécessaires pour exécuter un chemin donné est une opération difficile, et le nombre de chemin croît exponentiellement avec le nombre d'instructions conditionnelles présentes, ce que le rend difficilement applicable dans le cas de très gros codes embarqués.

Insuffisance du test. Même avec le meilleur critère de couverture possible, le test d'un programme ne permet malheureusement pas de s'assurer de l'absence de bugs[1]. Dans l'absolu, seul

[1] *"Program testing can be used to show the presence of bugs, but never to show their absence !"*, citation de Dijkstra extraite de "Notes On Structured Programming"

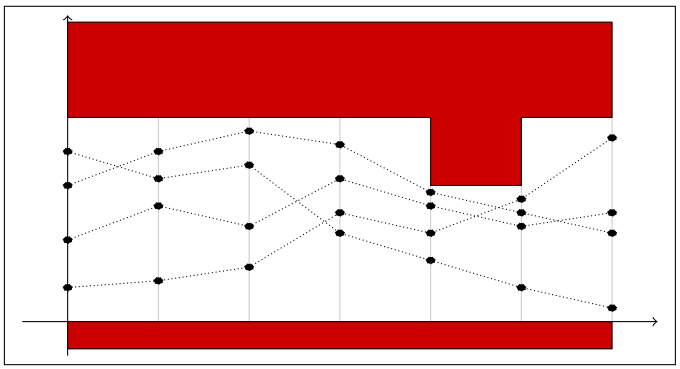


FIG. 1.2 – Exemple d'exécution d'un jeu de test comportant quatre valeurs.

un test exhaustif du programme avec chaque valeur possible pour les variables d'entrées permettrait de prouver sa correction (le test exhaustif peut ainsi être vu comme une technique, certes rudimentaire, de preuve de programmes). Pour des programmes de taille industrielle, ceci n'est pas envisageable : pour un programme de 1.000.000 de lignes de code contenant 50.000 variables flottantes codées sur 64 bits (soit la taille des programmes embarqués dans les derniers avions d'Airbus [DS07]), le nombre d'états possibles pour l'ensemble du programme est d'environ 10^{27}, et il est donc impossible de tous les tester. Concrètement, il restera toujours des cas particuliers que les différents jeux de tests n'auront pas pris en compte et qui peuvent se révéler dangereux. De plus, les techniques de test sont difficilement automatisables : si le test structurel s'intéresse aux exécutions réelles du programme, il est nécessaire d'avoir des spécifications permettant de juger la validité des sorties obtenues pour chaque jeu de test. L'automatisation de cette tâche requiert l'utilisation d'un *oracle*, ce qui pose entre autre le problème de la confiance portée dans cet oracle.

Nous laisserons le mot final à l'équipe d'enquête chargée de déterminer les causes de l'explosion d'Ariane 5, et qui en conclusion de son rapport ([Boa96], page 12) affirme ceci :
"The extensive reviews and tests carried out during the Ariane 5 development programme did not include adequate analysis and testing of the inertial reference system or of the complete flight control system, which could have detected the potential failure."

1.2.3 Analyse statique : vérification statique

Contrairement au test, l'analyse statique veut être exhaustive : le but est de trouver statiquement (c'est-à-dire sans l'exécuter) des propriétés qui soient vraies pour toutes les exécutions du programme. Ces propriétés peuvent être de différentes natures. Les plus simples sont sûrement les propriétés de sûreté (*safety*), où l'on cherche des invariants sur les valeurs des variables du programme, c'est-à-dire une plage de valeurs $X^\sharp$ telle qu'au cours de toutes les exécutions du programme, la variable x reste dans $X^\sharp$. La forme choisie pour représenter $X^\sharp$ (intervalles, octogones [Min01], polyèdres, [CH78]...) influe grandement sur la précision de cette analyse. Un tel invariant permet par exemple de montrer qu'il n'y a pas de RTE due à une division par zéro : si $0 \notin X^\sharp$, alors la variable x ne pourra pas causer de division par zéro. Un autre type de propriétés est l'absence de RTE due à un buffer overflow. Cela intervient, par exemple, lorsque l'on essaie de copier une chaîne de caractères de longueur n dans une chaîne de longueur $m < n$. Pour ce genre de propriétés, l'utilisation des algèbres $max - plus$ s'est révélée très utile [AGG08]. Citons enfin les propriétés numériques des programmes : une mesure utile de la qualité de l'implémentation d'un algorithme est souvent l'erreur de calcul du programme due à l'utilisation des nombres flottants [PGM03]. En effet, comme nous l'avons vu dans l'exemple du missile Patriot, les nombres flottants sont censés être une représentation des nombres réels mais sur un nombre fini de bits. Cela introduit naturellement un biais, et les calculs effectués dans l'arithmétique des nombres flottants [P7585] diffèrent, parfois grandement, des calculs effectués dans l'arithmétique réelle. Mesurer cette erreur peut permettre de corriger certaines erreurs de conception et/ou d'implémentation d'un algorithme. Les

propriétés de sûreté permettent donc de prouver qu'aucun mauvais comportement ne va arriver lors de l'exécution du programme.

Des propriétés d'un autre type peuvent aussi être intéressantes à prouver. Ainsi, la terminaison du programme est souvent une question importante [CPR06a], de même que la détection de code mort (c'est-à-dire une portion du programme qui ne sera jamais exécutée). Dans le cadre des programmes réactifs, on cherche également à avoir des garanties quant à la réponse du programme à un signal quelconque. De telles propriétés sont souvent classées dans la catégorie des propriétés de vivacité (*liveness*) [CPR06b], et sont considérées comme plus difficiles à prouver : il s'agit dans ce cas de prouver que le programme aura un bon comportement.

Clairement, l'ensemble des exécutions d'un programme est, sinon infini, de très grande taille, et les propriétés exactes mentionnées ci-dessus sont incalculables (elles nécessiteraient pour la plupart le calcul d'un point-fixe par une itération infinie). Pour corser le tout, tous ces problèmes (à savoir quelle est l'erreur maximale de calcul, y a-t-il une division par zéro,...) sont indécidables dans le cas général. Cela revient à dire que l'on ne peut pas trouver d'algorithme capable de répondre correctement pour tout programme P à la question suivante : y-a-t-il une exécution de P qui mène à une division par zéro ? Pour contourner ce problème, la théorie de l'interprétation abstraite [CC77, CC92a] propose de construire une méthode qui, à la même question, répondra OUI, NON ou PEUT-ETRE. Si la méthode répond OUI, alors il y aura effectivement une division par zéro lors d'une des exécutions du programme. Si la méthode répond NON, alors le programme est sûr (au moins du point de vue des divisions par zéro). Enfin, si la méthode répond PEUT-ETRE, c'est que nous n'avons pas été en mesure de prouver l'un ou l'autre des deux cas. C'est ce qu'on appelle une fausse alarme : il est possible qu'une des exécutions du programme produise une division par zéro, mais nous n'avons pas été capable de l'affirmer ou de l'infirmer. Nous ne calculerons donc plus la propriété exacte mais une abstraction de cette propriété, en imposant une contrainte de sûreté : la propriété abstraite calculée ne doit oublier aucune exécution concrète. Ainsi, si la réponse de la méthode est NON pour le programme P, alors on doit pouvoir affirmer qu'aucune exécution de P ne provoque de division par zéro. Pour calculer ces propriétés abstraites, nous utiliserons des domaines abstraits, c'est-à-dire des domaines permettant de représenter de manière finie un ensemble infini de valeurs (et donc un ensemble infini d'exécutions du programme), et nous *interpréterons* le programme sur ce domaine abstrait. Ainsi, chaque opérateur syntaxique du langage considéré sera traduit en un équivalent abstrait opérant sur une valeur abstraite. Nous pourrons donc calculer un ensemble potentiellement infini de comportements possibles. La contrainte de sûreté se traduit au niveau des opérateurs de la manière suivante : si op est un opérateur unaire du langage, X un ensemble de valeurs et $X^\sharp$ une valeur abstraite contenant au moins tout X, et si $op^\sharp$ est l'opérateur abstrait correspondant, alors il faut que $op^\sharp(X^\sharp)$ contienne au moins $\{op(x) \ : \ x \in X\}$. Enfin, pour répondre au problème de l'incalculabilité des invariants, nous utiliserons des techniques d'accélération de la convergence qui permettent de rendre le calcul des itérations de point fixe ultimement stationnaire. Nous détaillerons ces techniques dans le chapitre 2.

Si l'on reprend l'exemple de la figure 1.2, l'interprétation abstraite consiste à calculer à chaque ligne du code une surapproximation de l'ensemble des valeurs prises par x en cette ligne lors de toutes les exécutions du programme. Cela correspond aux rectangles bleus sur la figure 1.3. On voit bien que ces rectangles contiennent l'ensemble des valeurs calculées par le test, ce qui est imposé par les contraintes de sûreté, et même plus. Ainsi, au point de contrôle 5, la valeur abstraite calculée (rectangle bleu) intersecte la zone dangereuse (rectangle rouge), alors qu'aucune des exécutions du programme n'entre dans la zone dangereuse. Nous avons donc affaire à une fausse alarme : la perte de précision due à l'utilisation d'un domaine abstrait nous empêche de conclure quant à la sûreté du programme. L'utilisation d'un domaine abstrait plus précis pourrait lever cette fausse alarme.

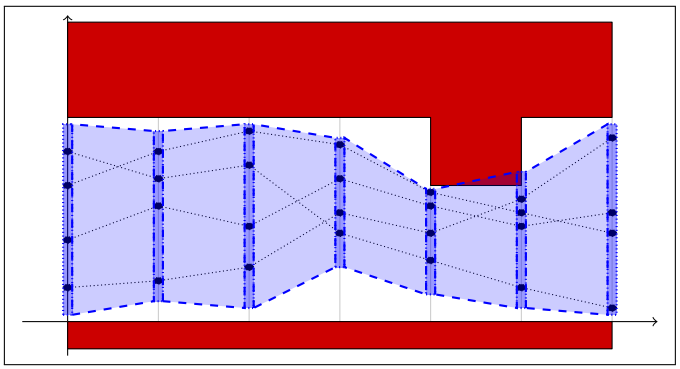


FIG. 1.3 – Vérification par interprétation abstraite.

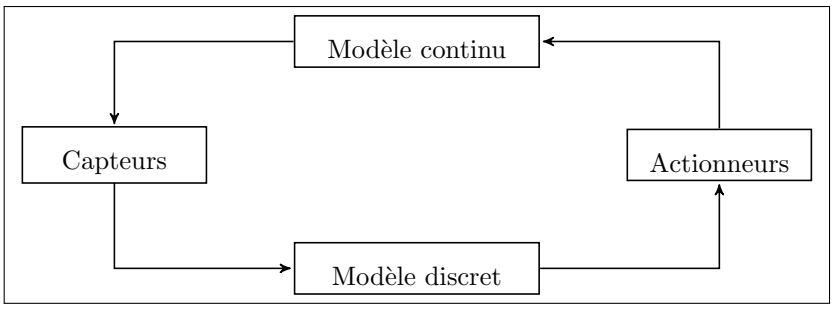


FIG. 1.4 – Mode de fonctionnement d'un système hybride discret/continu.

1.3 Pour aller plus loin

1.3.1 Du modèle hybride au modèle discret

Si l'on regarde de plus près le cycle en V pour le développement de programmes embarqués (figure 1.1), on remarque une différence fondamentale entre la spécification du système et le développement des composants : le système que l'on spécifie à haut niveau est un système profondément hybride, alors que le développement ne prend en compte que ses sous parties discrètes. Le terme de systèmes hybrides [Ant00, SS00] est ici employé pour désigner des systèmes comportant des éléments discrets (c'est-à-dire dont l'évolution suit un temps discret) et des éléments continus (c'est-à-dire dont l'évolution suit le temps continu classique). L'immense majorité des programmes embarqués sont de tels systèmes : ils reçoivent en entrée des valeurs continues, fournies par des capteurs et agissent sur l'environnement physique qui les entoure au moyen d'actionneurs (figure 1.4). Généralement, la spécification de ce système, par exemple en Simulink, comprend les sous parties discrètes et continues. On dispose donc de beaucoup d'informations sur le système continu, informations que l'on n'utilise généralement pas lors de la vérification des parties discrètes. En effet, la technique classique permettant d'analyser ces programmes embarqués est d'abstraire le monde continu (c'est-à-dire les entrées du programme) par un intervalle, par exemple la plage de valeurs que le capteur peut fournir.

Cette surapproximation de l'environnement continu par un intervalle constant dans le temps est une source importante d'imprécision des analyses actuelles de programmes embarqués [GMP06]. Considérons par exemple un programme qui intègre, en utilisant la méthode des rectangles, des valeurs provenant d'un capteur. Supposons également que ce capteur mesure une variable continue dont l'évolution suit la fonction $f(t)$, qui vérifie : $\forall t \in \mathbb{R}_+,\ f(t) \in [0, 2]$. Clairement, l'analyse statique utilisera comme abstraction des entrées l'intervalle $[0, 2]$. De fait, la meilleur abstraction possible pour la variable d'intégration est $[0, 2 \cdot n \cdot h]$, où n est le nombre d'itérations dans le calcul d'intégration et h le pas d'intégration. Cette valeur est une surapproximation assez grossière

de la réalité : clairement, la variable d'intégration a une valeur proche de $\int_0^{n\times h} f(s)ds$ après n itérations. Limiter cette surapproximation est un des enjeux de cette thèse. Par ailleurs, certaines propriétés de sûreté intéressantes pour les programmes embarqués ne sont calculables que si on prend en considération l'environnement. Prenons l'exemple de la batterie antimissile Patriot. La non-interception du missile Scud est due à une perte de précision lors de la conversion d'un entier vers un nombre flottant codé sur 24 bits [GAO92]. Une analyse statique du programme avec un outil comme Fluctuat [GMP02] permet de déceler cette perte de précision, et fournit comme résultat une surapproximation de l'erreur commise. Cependant, cette information ne permet pas de décider de la sûreté du système antimissile Patriot sans connaître des informations supplémentaires comme la vitesse de croisière des missiles Scud ou l'intervalle de temps entre deux prises de décision du missile Patriot. Ainsi, la propriété la plus intéressante (la batterie Patriot va-t-elle intercepter les missiles Scud) ne peut-être prouvée sans une connaissance (et donc une analyse) du système dans son ensemble. Inversement, on peut très bien imaginer un système pour lequel l'erreur numérique commise par le programme est très importante mais qui reste sûr car il prend malgré tout les bonnes décisions (c'est-à-dire les mêmes que celles prises par un système parfait qui ne ferait aucune erreur de calcul).

Nous voyons donc qu'une analyse du système hybride dans son ensemble permet à la fois d'améliorer la précision de l'analyse de la partie discrète mais aussi d'étendre l'ensemble des propriétés que l'on peut démontrer. C'est cette observation qui a motivé le travail de cette thèse.

1.3.2 Analyse du système hybride

Pour étendre les capacités d'analyse sur des programmes embarqués, il faut donc étudier le système hybride dans lequel ils sont plongés. Nous voulons appliquer aux systèmes hybrides les techniques de validation (c'est-à-dire l'analyse statique par interprétation abstraite) qui ont fait leur preuve pour la validation des programmes discrets. Comme le montre la figure 1.1, il existe deux moyens de faire cela : soit en faisant remonter la vérification plus haut dans le cycle de développement en essayant de valider des modèles de haut niveau, soit en faisant descendre le caractère hybride au niveau du code pour ajouter de l'information aux analyses existantes.

Les automates hybrides [Hen96] et les autres modèles hybrides de haut niveau (tels que le ϕ-calcul ou Hybrid-CC) correspondent à la première approche. Les automates hybrides sont une extension naturelle des automates temporisés [AD94] dans laquelle on autorise les variables à avoir des variations temporelles définies par des équations différentielles quelconques. Un automate hybride comprend donc plusieurs états, chaque état étant associé à une équation de flot qui décrit l'évolution des variables continues dans cet état. Une transition dans l'automate représente l'effet d'un actionneur sur le système et entraîne donc un changement dans sa dynamique. Nous détaillerons le formalisme des automates hybrides et les techniques permettant leur vérification (notamment par *model checking*) dans la section 4.2.

Cette approche qui consiste à remonter les techniques de vérification au plus haut du cycle de développement est clairement nécessaire, elle n'est cependant pas suffisante. En effet, une garantie sur le modèle n'est que difficilement traduisible en une garantie sur le code source, tout comme une garantie sur le code n'est pas une garantie sur le binaire. Ainsi, il est important d'appliquer les techniques de vérification à tous les niveaux du cycle de développement. De plus, un certain nombre de propriétés ne peuvent être démontrées que sur le code source, et pas sur le modèle de haut niveau (la stabilité numérique ou l'absence de RTE par exemple). Il nous semble donc important de continuer la vérification au niveau du programme embarqué, mais en ajoutant des informations concernant l'environnement physique dans lequel le programme est exécuté pour améliorer la précision des invariants et le type de propriétés que l'on peut calculer. Cette thèse présente une méthodologie pour cela.

1.3.3 Autre type de propriétés hybrides

Remarquons pour finir que prendre en compte l'environnement continu n'est pas la seule possibilité d'extension de l'analyse des programmes embarqués. De nombreux travaux visent en effet à

ajouter à la vérification des programmes de l'information concernant l'architecture matérielle sur laquelle le programme est exécuté. Cela revient à considérer un autre type de modèles hybrides, le terme hybride désignant désormais des systèmes ayant une partie logicielle et une partie matérielle. Comme pour les systèmes hybrides discret/continu, cet ajout d'une partie matérielle vise d'une part à améliorer les propriétés calculées par les techniques classiques d'analyse statique [NGC08] et d'autre part à prouver des propriétés nouvelles. Ainsi, l'analyse du pire temps d'exécution d'un programme [FHL+01, HF05] ne peut être effectuée de manière suffisamment précise que si l'on prend en compte l'environnement matériel associé au logiciel. De même, les analyses de précision numérique supposent que les opérations sont conformes à la norme IEE754, ce qui est une hypothèse sur le matériel sur lequel le programme s'exécutera.

1.4 Contributions

Les contributions de cette thèse sont de trois ordres.

Un modèle et une sémantique dénotationnelle pour le code hybride Nous présentons un modèle de systèmes hybrides [BM08b] qui étend les langages de programmation impératifs classiques dans le but d'intégrer à la phase de développement du cycle en V des aspects hybrides.

Ce modèle a été développé avec les objectifs suivants : il doit être le moins intrusif et le plus proche possible du code pour pouvoir être utilisé efficacement sur des programmes embarqués existants, et il doit être conçu pour empêcher les phénomènes hybrides physiquement impossibles de se réaliser. Ce double objectif a été atteint. Dans le modèle que nous proposons, un système hybride est un programme qui compose le sous-système discret, couplé à un sous-système continu modélisé par un ensemble fini d'équations différentielles. Le programme est écrit dans une extension d'un langage impératif (comme le C) ; l'extension consiste en trois nouveaux mots clés qui permettent de modéliser l'action des capteurs, des actionneurs ainsi que l'avancement du temps. Le fait que le temps soit entièrement dirigé par la partie discrète du système hybride est sûrement l'aspect qui différencie le plus ce modèle des automates hybrides. C'est également pour cela que notre modèle interdit des comportements physiquement impossible comme l'effet Zénon, qui consiste à effectuer en un temps fini une infinité d'actions discrètes. Ce phénomène est largement étudié pour les modèles hybrides classiques [ZJLS01] et nous montrons ici qu'il est facile de vérifier qu'un système hybride écrit dans notre modèle ne présente pas de comportements Zénon. Le sous-système continu est modélisé par un ensemble d'équations différentielles qui représente les différents modes continus correspondant aux différentes actions possibles du système discret (comme ouvrir/fermer une vanne, allumer/éteindre le chauffage, etc.).

Nous proposons une sémantique dénotationnelle pour ce modèle. Cette sémantique doit calculer l'évolution du système hybride, c'est-à-dire à la fois l'évolution discrète (i.e. le résultat du programme) et l'évolution continue (i.e. les solutions des équations différentielles). Pour construire cette sémantique, nous sommes partis du constat qu'une équation différentielle et un programme décrivent de manière similaire un système dynamique : tous deux définissent l'évolution du système en liant l'état suivant du système à son état courant, que ce soit dans le cas discret pour un programme ou dans le continu pour une équation différentielle. Notre sémantique est construite en trois étapes. Dans un premier temps, nous supposons que l'évolution discrète est parfaitement connue, et nous calculons l'évolution continue. Pour cela, nous exprimons la solution d'une équation différentielle comme le point fixe (au sens de Tarski) d'un opérateur monotone défini sur le treillis des fonctions continues à valeur dans les intervalles. Nous montrons que ce point fixe, calculable par les itérées de Kleene, est la solution de l'équation différentielle. Dans un second temps, nous supposons à l'inverse que l'évolution continue du système est connue, c'est-à-dire que l'on connaît les solutions des équations différentielles. Nous calculons alors la sémantique discrète de manière classique en définissant des dénotations aux mots-clés supplémentaire introduits par notre modèle. Enfin, nous définissons la sémantique du modèle hybride comme une combinaison des deux sémantiques précédentes qui calcule en même temps les évolutions continues et discrètes.

Une analyse statique par interprétation abstraite du système hybride Nous posons ensuite les bases d'une analyse globale du système hybride. En supposant que l'on sait calculer une abstraction des solutions d'une équation différentielle, nous montrons comment une pré-analyse du programme embarqué, indépendemment de son environnement, permet de construire une abstraction des effets de la partie discrète sur l'environnement continu. Nous construisons pour cela un *octree* abstrait, c'est-à-dire un arbre régulier permettant de détecter facilement les zones de l'espace continu dans lesquelles le programme va modifier la dynamique. Cette analyse fait le lien entre les techniques d'analyse par intervalle pour construire des ensembles atteignables non convexes et une analyse d'accessibilité dans les programmes impératifs basée sur une interprétation abstraite classique. Des premiers résultats expérimentaux montrent l'intérêt d'une telle approche qui, par une séparation entre l'analyse de la partie discrète et de la partie continue, simplifie grandement l'analyse du système hybride.

Un outil d'intégration garantie par la méthode de Runge-Kutta Nous présentons enfin une nouvelle technique d'intégration garantie d'équations différentielles ordinaires [BM06a, BM07a]. L'intégration garantie consiste à calculer des bornes encadrant la solution d'un équation différentielle. Les méthodes classiques pour cela sont construites à partir d'un développement en série de Taylor de la fonction à intégrer et une arithmétique d'intervalle. La plupart de ces méthodes souffrent du *wrapping effect*, c'est-à-dire un grossissement des bornes des intervalles. De plus, étant très éloignées des méthodes numériques classiques, elles ne bénéficient pas de l'expertise que possèdent les numériciens sur la résolution approchée d'équations différentielles. Nous avons développé une méthode qui essaie de contourner le problème du *wrapping effect* en séparant valeurs flottantes et intervalles, comme cela a été fait pour l'analyse statique de programmes numériques [Mar05], et qui repose sur une méthode numérique classique pour pouvoir bénéficier de l'expérience des numériciens. Notre méthode repose sur l'algorithme numérique classique de Runge-Kutta RK4 et nous montrons qu'il est possible de calculer de manière garantie des bornes sur l'erreur globale commise par l'intégration numérique.

Cette erreur est exprimée comme la somme de trois termes. Tout d'abord, l'erreur numérique due à l'implémentation de RK4 sur un ordinateur qui ne dispose que d'une précision finie pour les calculs est prise en compte en utilisant le domaine de l'erreur globale qui a été initialement développé pour la validation de programmes numériques [Mar05]. Ensuite, l'erreur due à la méthode elle-même est exprimée. Cette erreur apparaît car la méthode RK4 n'est que d'ordre 5, et nous utilisons le développement en série de Taylor avec reste de Lagrange pour la calculer. Nous reprenons et étendons ici les travaux de Bieberbach [Bie51]. Enfin, nous calculons la propagation de l'erreur d'un pas de calcul à l'autre. Cette erreur est exprimée grâce au théorème des valeurs intermédiaires, nous utilisons ici les travaux de Carr [Car58].

Cette méthode a été implémentée dans une librairie d'intégration garantie, GRKLib. Cette librairie C++ permet de calculer des encadrements garanties des solutions d'équations différentielles représentées sous la forme d'une fonction C++. Nous présenterons des comparaisons entre cette librairie et les outils existants, notamment VNODE [NJ99].

1.5 Plan de la thèse

Cette thèse comprend 3 parties distinctes.

Dans un premier temps nous ferons un état de l'art détaillé de plusieurs domaines.

Les chapitres 2 et 3 concernent les techniques d'analyse de la partie discrète et continue d'un système hybride. Au chapitre 2, nous formalisons le concept d'interprétation abstraite et donnons quelques exemples de domaines abstraits utilisés pour la vérification de programmes. Au chapitre 3, nous expliquons les concepts de l'intégration garantie d'équations différentielles ordinaires et décrivons les techniques classiques par méthode de Taylor.

Au chapitre 4, nous formalisons le concept d'automates hybrides et introduisons le vocabulaire relatif aux systèmes hybrides. Nous nous intéressons également aux techniques de vérification

relatives aux automates hybrides ainsi qu'aux autres modèles de systèmes hybrides.

Dans un deuxième temps, nous décrivons notre nouveau modèle pour les systèmes hybrides. Cette partie est constituée du seul chapitre 5 qui présente ce modèle et donne sa sémantique dénotationnelle. Nous ferons notamment quelques rappels sur les concepts de base de la théorie des domaines de Scott.

La troisième partie présente une technique d'analyse par interprétation abstraite de ces systèmes hybrides.

Au chapitre 6, nous montrerons que l'intégration garantie peut s'intégrer dans la théorie de l'interprétation abstraite. Nous définirons notamment le domaine des fonctions en escalier à valeur dans les intervalles. De plus, nous présenterons l'algorithme et la libraire GRKLib. Nous montrerons comment il est possible d'encadrer l'erreur globale due à l'utilisation d'une méthode numérique d'intégration quelconque, puis nous appliquerons cela à la méthode de Runge-Kutta RK4. Enfin, au chapitre 7, nous décrirons comment on peut analyser le système hybride dans son ensemble en séparant les analyses de la partie discrète et de la partie continue.

Nous terminerons sur le chapitre 8 qui résume les travaux de cette thèse et ouvre sur de nombreuses perspectives.

Le chapitre 5 a fait l'objet de l'article [BM08b], le chapitre 6 a fait l'objet des article [BM06b], [BM07a] et [BM08a]. Par ailleurs, ce travail de thèse a été présenté sous forme d'un poster à la conférence *Hybrid Systems : Computation and Control* [BM07b].

1.6 Exemple : les deux réservoirs.

La littérature sur les systèmes hybrides fournit d'innombrables exemples de systèmes plus ou moins complexes. Le plus connu est sûrement le système des deux réservoirs, qui sert généralement de référence pour les outils de vérification de systèmes hybrides [KSF+99]. Nous utiliserons dans cette thèse ce système pour illustrer les concepts relatifs aux problèmes hybrides.

Le système se présente sous la forme de deux réservoirs (figure 1.6) ; le réservoir supérieur sera appelé réservoir 1, le second sera le réservoir 2. Les deux réservoirs sont reliés entre eux par un tuyau horizontal, situé au fond du réservoir 1 et à une hauteur H par rapport au bas du réservoir 2. Le réservoir 2 possède également un tuyau d'évacuation situé à la hauteur 0. Nous noterons h_1 (resp. h_2) la hauteur de liquide dans le réservoir 1 (resp. dans le réservoir 2). Le réservoir 1 est rempli par un débit constant i, si bien que l'évolution des hauteurs h_1 et h_2 est donnée par l'équation différentielle (1.1), quand les deux tuyaux sont ouverts.

$$\begin{aligned} \dot{h_1} &= \begin{cases} i - k_1\sqrt{h_1} \text{ si } h_2 \leq H \\ i - k_1\sqrt{h_1 - h_2 + H} \text{ si } h_2 > H \end{cases} \\ \dot{h_2} &= \begin{cases} k_1\sqrt{h_1} - k_2\sqrt{h_2} \text{ si } h_2 \leq H \\ k_1\sqrt{h_1 - h_2 + H} - k_2\sqrt{h_2} \text{ si } h_2 > H \end{cases} \end{aligned} \tag{1.1}$$

Nous introduisons un comportement hybride à ce système de la manière suivante : nous supposerons que chaque tuyau horizontal possède une vanne (v_1 pour le tuyau reliant le réservoir 1 au réservoir 2, v_2 pour le tuyau d'évacuation) qui peut être ouverte ou fermée par un contrôleur. Sur la figure 1.6, la vanne v_1 est fermée et la vanne v_2 est ouvert. Nous supposerons de plus que chaque réservoir possède deux limites (l_1 et L_1 pour le réservoir 1, l_2 et L_2 pour le réservoir 2) au delà desquelles le système est en état critique (cela peut signifier que le réservoir déborde, ou qu'il est trop vide). Il faut donc qu'à tout instant, h_i vérifie $l_i \leq h_i \leq L_i$ pour $i = 1, 2$. Le contrôleur sert

à faire respecter cette propriété en ouvrant et/ou fermant les vannes quand cela est nécessaire. Le contrôleur possède donc quatre états discrets, suivant que v_1 et v_2 sont ouvertes ou fermées, et passera de l'un à l'autre pour contrôler les niveaux d'eau. Ces quatre états discrets correspondent à quatre évolutions continues décrites par les équations (1.2) à (1.5) de la figure 1.5. Nous ferons enfin une supposition supplémentaire, assez classique pour un système hybride, qui est que les actions décidées par le contrôleur mettent deux secondes à se réaliser. Ainsi, entre le moment où le contrôleur décide de fermer la vanne v_1 et le moment où celle-ci est effectivement fermer, il se sera écoulé deux secondes. Ainsi, le contrôleur devra anticiper l'évolution du niveau d'eau pour s'assurer que l'invariant $l_i \leq h_i \leq L_i$ est vérifié avec cette contrainte de délai supplémentaire.

v_1 ouverte , v_2 ouverte

$$\begin{cases} \dot{h_1} = \begin{cases} i - k_1\sqrt{h_1} \text{ si } h_2 \leq H \\ i - k_1\sqrt{h_1 - h_2 + H} \text{ si } h_2 > H \end{cases} \\ \dot{h_2} = \begin{cases} k_1\sqrt{h_1} - k_2\sqrt{h_2} \text{ si } h_2 \leq H \\ k_1\sqrt{h_1 - h_2 + H} - k_2\sqrt{h_2} \text{ si } h_2 > H \end{cases} \end{cases} \quad (1.2)$$

v_1 ouverte , v_2 fermée

$$\begin{cases} \dot{h_1} = \begin{cases} i - k_1\sqrt{h_1} \text{ si } h_2 \leq H \\ i - k_1\sqrt{h_1 - h_2 + H} \text{ si } h_2 > H \end{cases} \\ \dot{h_2} = \begin{cases} k_1\sqrt{h_1} \text{ si } h_2 \leq H \\ k_1\sqrt{h_1 - h_2 + H} \text{ si } h_2 > H \end{cases} \end{cases} \quad (1.3)$$

v_1 fermée , v_2 ouverte

$$\begin{cases} \dot{h_1} = i \\ \dot{h_2} = -k_2\sqrt{h_2} \end{cases} \quad (1.4)$$

v_1 fermée , v_2 fermée

$$\begin{cases} \dot{h_1} = i \\ \dot{h_2} = 0 \end{cases} \quad (1.5)$$

FIG. 1.5 – Équations différentielles pour le système des deux réservoirs.

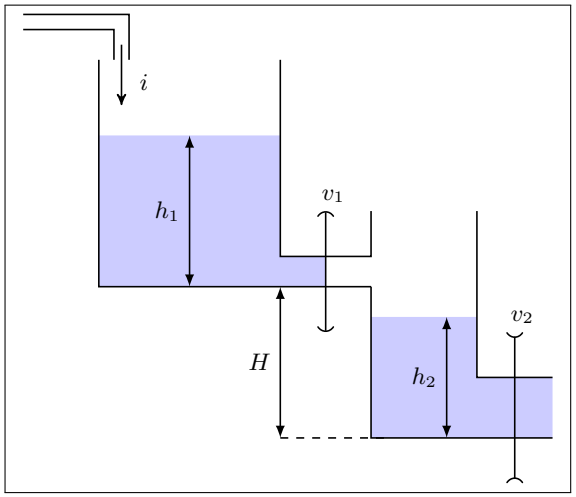


FIG. 1.6 – Système des deux réservoirs.

1.7 Notations

Valeurs. L'ensemble des nombres réels sera noté $\mathbb{R}$, les réels positifs $\mathbb{R}_+ = \{x \in \mathbb{R} \ : \ x \geq 0\}$ et les réels négatifs $\mathbb{R}_-$. Les nombres naturels seront notés $\mathbb{N}$ et les nombres flottants $\mathbb{F}$. Les nombres flottants seront écrits en gras : $\boldsymbol{x} \in \mathbb{F}$.

Intervalles. Soit un domaine de valeurs $\mathbb{D}$ muni d'un ordre total $\leq_{\mathbb{D}}$. Pour $a, b \in \mathbb{D}$, on notera $[a, b] = \{x \in \mathbb{D} \ : \ a \leq_{\mathbb{D}} x \leq_{\mathbb{D}} b\}$ l'intervalle contenant tout les éléments de $\mathbb{D}$ compris entre a et b. Clairement, si $b < a$, on a $[a, b] = \emptyset$. Nous noterons $I_{\mathbb{D}}$ l'ensemble des intervalles de $\mathbb{D}$:

$$I_{\mathbb{D}} = \{[a, b] \ : \ a \in \mathbb{D}, \ b \in \mathbb{D}\}$$

En particulier, nous disposerons des intervalles réels $I_{\mathbb{R}}$ et flottants $I_{\mathbb{F}}$. Les éléments de $I_{\mathbb{D}}$ seront notés entre crochets : $[x] \in I_{\mathbb{D}}$. Pour un élément $[x] \in I_{\mathbb{D}}$, nous noterons $\underline{x}$ et $\overline{x}$ ses bornes inférieures et supérieures, respectivement. Ainsi, $[x] = [\underline{x}, \overline{x}]$. Nous utiliserons souvent l'arithmétique d'intervalles définie dans [Moo66, Moo79] et nous utiliserons, par abus de notations, les mêmes symboles $(+, \times, \ldots)$ pour l'arithmétique réelle, flottante et d'intervalle.

Vecteurs et matrices. Pour un domaine de valeurs $\mathbb{D}$, nous noterons $\mathbb{D}^n$ l'ensemble des vecteurs de dimension n dont les éléments sont des éléments de $\mathbb{D}$:

$$\mathbb{D}^n = \left\{\left(x_1, \ldots, x_n\right) \ : \ \forall i \in [\![1, n]\!], \ x_i \in \mathbb{D}\right\}$$

Un élément de $\mathbb{D}^n$ sera noté $\vec{x}$; ainsi, les vecteurs d'intervalles réels seront notés $[\vec{x}] \in I_{\mathbb{R}}^n$ et les vecteurs de nombres flottants seront notés $\vec{\boldsymbol{x}} \in \mathbb{F}^n$. Enfin, l'ensemble des matrices de dimension $n \times m$ (n lignes, m colonnes) sera noté $\mathbb{D}^{(n,m)}$ et ses éléments seront écrits avec une majuscule : $\boldsymbol{A} \in \mathbb{F}^{(n,m)}$.

Fonctions. Nous noterons $\mathcal{C}^0(D)$ l'ensemble des fonctions continues de $\mathbb{R}$ dans un espace métrique D. Pour définir une fonction, nous utiliserons parfois la notation $\lambda x.g(x)$: lorsque l'on écrit $f = \lambda x.g(x)$, cela signifie que pour tout x du domaine de définition de f, on a $f(x) = g(x)$.

Première partie

État de l'art

La première partie de cette thèse vise à présenter l'état de l'art quant à l'analyse et la vérification des systèmes hybrides. Dans un premier temps, nous étudierons les techniques classiques utilisées pour l'analyse des sous-parties discrètes et continues de ces systèmes. Nous insisterons ainsi sur les techniques d'analyse statique par interprétation abstraite (chapitre 2) pour la partie discrète, et sur les techniques d'intégration garantie d'équations différentielles (chapitre 3) pour la partie continue. Ensuite, nous présenterons les modèles classiques permettant de représenter les systèmes hybrides dans leur ensemble (chapitre 4). Nous insisterons particulièrement sur les *automates hybrides* (section 4.2) et évoquerons les techniques de vérification sur ces modèles. Cela nous mènera aux problèmes d'*accessibilité* et d'*indécidabilité* (section 4.4). Des modèles plus récents construits comme une extension des calculs de processus seront également présentés (section 4.3).

CHAPITRE 2

Analyse de la partie discrète : interprétation abstraite

Comme nous l'avons mentionné en introduction, nous nous intéressons à des systèmes hybrides dont la partie discrète est un programme et dont la partie continue est l'environnement physique avec lequel le programme interagit. Si le test est la technique la plus utilisée pour vérifier le comportement d'un programme, l'analyse statique des programmes vise à combler les lacunes du test en apportant des preuves du bon (ou mauvais) comportement de ces programmes quelles que soient les valeurs prises par les variables d'entrée, et cela sans exécuter le programme. Les méthodes les plus basiques ont recourt à une analyse purement syntaxique des programmes en cherchant des motifs qui sont connus pour être potentiellement dangereux. Ces techniques, connues sous le nom de *lint* [Joh77], sont d'avantage destinées à trouver des bugs qu'à prouver leur absence. Pour obtenir un telle preuve, il est nécessaire d'étudier non seulement la syntaxe d'un programme mais aussi, et surtout, sa sémantique. La sémantique d'un programme est une description formelle de sa signification, c'est-à-dire de son comportement attendu. On se doute donc bien que les outils d'analyse reposant sur une vision sémantique du programme sont potentiellement bien plus puissants que ceux qui ont une vision purement syntaxique. Ce chapitre décrit une de ces techniques d'analyse statique, l'interprétation abstraite [CC77, CC92a]. Nous commençons par rappeler quelques bases concernant la théorie des domaines (section 2.1), puis nous définissons formellement la sémantique d'un langage de programmation à travers l'exemple d'un langage impératif jouet (section 2.2). Enfin, nous décrivons la théorie de l'interprétation abstraite (section 2.3) qui permet de calculer une surapproximation de l'ensemble des comportements d'un programme pour un ensemble de valeurs d'entrée.

2.1 Rappels sur la théorie des domaines

Les notions d'ensembles ordonnés, de fonctions continues et de point fixe sont au cœur de la théorie de la sémantique des langages de programmation et de l'interprétation abstraite. Nous rappelons ici les définitions et résultats de base permettant de comprendre la suite de ce chapitre. Nous commençons par la notion d'ordre partiel, de borne inférieure et supérieure, puis nous introduisons le concept d'ordre partiel complet et de treillis. Nous terminerons enfin par la définition d'une fonction continue et nous donnerons les théorèmes d'existence et de calcul de point fixe. Pour chaque définition, nous apporterons un exemple explicatif.

2.1.1 Ordre partiel

Définition 2.1 (Ordre partiel) Un ordre partiel est un ensemble P muni d'une relation binaire $\sqsubseteq$ qui est :

1. *réflexive*, $\forall x \in P,\ x \sqsubseteq x$;
2. *transitive*, $\forall x, y, z \in P,\ x \sqsubseteq y \wedge y \sqsubseteq z \Rightarrow x \sqsubseteq z$;
3. *antisymétrique*, $\forall x, y \in P,\ x \sqsubseteq y \wedge y \sqsubseteq x \Rightarrow x = y$.

Nous noterons l'ordre partiel $(P, \sqsubseteq)$.

Exemple 2.1 Dans tout ce chapitre, nous utiliserons l'ensemble ordonné $Sign$ défini ainsi :

$$Sign = \left\{\bot, \top, +, -, 0, \dot{+}, \dot{-}, \varnothing\right\};.$$

Cet ensemble contient donc 7 éléments qui ont les significations suivantes :

1. $\bot$ représente le plus petit élément ;
2. $\top$ représente le plus grand élément ;
3. 0 représente le nombre 0 ;
4. $-$ représente tous les nombres strictement négatifs ;
5. $+$ représente tous les nombres strictement positifs ;
6. $\dot{-}$ représente tous les nombres négatifs ou nul ;
7. $\dot{+}$ représente tous les nombres positifs ou nul ;
8. $\varnothing$ représente tous les nombres non nuls.

La relation d'ordre $\sqsubseteq$ sur $Sign$ est définie par :

$$\begin{array}{ccccccc} \bot \sqsubseteq - & \bot \sqsubseteq 0 & \bot \sqsubseteq + & - \sqsubseteq \dot{-} & - \sqsubseteq \varnothing & + \sqsubseteq \dot{+} & + \sqsubseteq \varnothing \\ & 0 \sqsubseteq \dot{-} & 0 \sqsubseteq \dot{+} & \dot{-} \sqsubseteq \top & \dot{+} \sqsubseteq \top & \varnothing \sqsubseteq \top & \end{array}$$

Graphiquement, nous représenterons cet ordre partiel par un *diagramme de Hasse* (figure 2.1), dans lequel chaque élément de $Sign$ est représenté par un nœud rouge et l'ordre est donné par des arêtes entre ces nœuds.

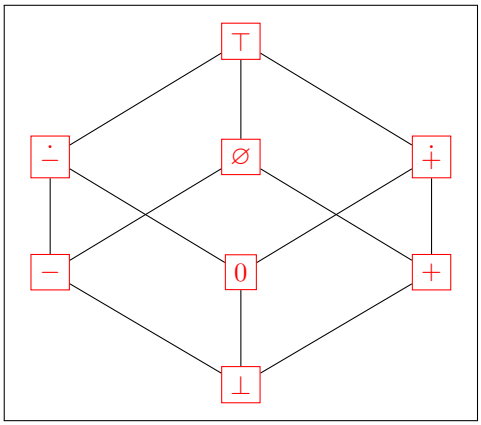


FIG. 2.1 – Diagramme de Hasse représentant l'ordre partiel de l'exemple 2.1.

Définition 2.2 (Borne supérieure) Soit $(P, \sqsubseteq)$ un ordre partiel, et $Q \subseteq P$. Un élément $p \in P$ est un majorant de Q si et seulement si

$$\forall x \in Q,\ x \sqsubseteq p\ . \tag{2.1}$$

On dira que p est la borne supérieure de Q si c'est le plus petit des majorants, c'est-à-dire que p est un majorant de Q et

$$\forall y \in P,\ \big(\forall x \in Q,\ x \sqsubseteq y\big) \Rightarrow p \sqsubseteq y\ . \tag{2.2}$$

Nous noterons $p = \bigsqcup Q$. Si l'ensemble Q possède deux éléments q_1, q_2, nous utiliserons la notation infixe : $p = q_1 \sqcup q_2$.

Exemple 2.2 Soit $S = \{-, +\}$ et $S' = \{\dot{-}, \dot{+}, 0\}$. Alors, on a $\bigsqcup S = \varnothing$ et $\bigsqcup S' = \top$.

Définition 2.3 (Borne inférieure) Soit $(P, \sqsubseteq)$ un ordre partiel, et $Q \subseteq P$. Un élément $p \in P$ est un minorant de Q si et seulement si

$$\forall x \in Q,\ p \sqsubseteq x\ . \tag{2.3}$$

On dira que p est la borne inférieure de Q si c'est le plus grand des minorants, c'est-à-dire que p est un minorant de Q et

$$\forall y \in P,\ \big(\forall x \in Q,\ y \sqsubseteq x\big) \Rightarrow y \sqsubseteq p\ . \tag{2.4}$$

Nous noterons $p = \bigsqcap Q$. Si l'ensemble Q possède deux éléments q_1, q_2, nous utiliserons la notation infixe : $p = q_1 \sqcap q_2$.

Exemple 2.3 Soit $S = \{-, +\}$ et $S' = \{\dot{-}, \dot{+}\}$. Alors, on a $\bigsqcap S = \bot$ et $\bigsqcap S' = 0$.

2.1.2 Ordre partiel complet, treillis

Définition 2.4 (Chaîne croissante) Soit $(P, \sqsubseteq)$ un ordre partiel. Une chaîne croissante est un sous-ensemble Q de P avec un plus petit élément et telle que :

$$\forall x, y \in Q,\ x \sqsubseteq y \vee y \sqsubseteq x \tag{2.5}$$

Autrement dit, une chaîne croissante est un sous-ensemble de P tel que les éléments de Q sont deux-à-deux comparables et tel que Q possède une borne inférieure.

Exemple 2.4 L'ensemble $S = \{\bot, -, \varnothing, \top\}$ est une chaîne croissante de $Sign$ dont $\bot$ est le plus petit élément. $S' = S \cup \{0\}$ n'est pas une chaîne croissante car 0 et $-$ ne sont pas comparables.

Définition 2.5 (Ordre partiel complet) Un ordre partiel $(P, \sqsubseteq)$ est dit *complet* (*complete partial order*, ou CPO) si et seulement si toute chaîne croissante non vide Q admet une borne supérieure.
Un CPO pointé (*pointed CPO*) est une CPO $(P, \sqsubseteq)$ muni d'un plus petit élément $\bot$ vérifiant $\forall p \in P,\ \bot \sqsubseteq p$. Dans la suite, le terme CPO désignera un CPO pointé.

Remarque Tout ordre partiel fini, ou de hauteur finie (c'est-à-dire ne possédant pas de chaîne croissante infinie), avec un plus petit élément est un CPO. En particulier, $Sign$ est un CPO.

Définition 2.6 (Treillis) Un treillis est un ordre partiel $(P, \sqsubseteq)$ tel que toute partie *finie* $Q \subseteq P$ admet une borne supérieure et une borne inférieure.

Exemple 2.5 L'ordre partiel *Sign* est clairement un treillis. Cependant, si on rajoute la règle $0 \sqsubseteq \varnothing$ pour l'ordre $\sqsubseteq$, on a toujours un CPO mais le nouvel ordre partiel n'est pas un treillis : si on prend $S = \{-, 0\}$, alors l'ensemble des majorants de S est $\{\top, \dot{-}, \varnothing\}$ qui ne possède pas de plus petit élément.

Définition 2.7 (Treillis complet) Un treillis complet est un ordre partiel $(P, \sqsubseteq)$ tel que tout ensemble $Q \subseteq P$ (fini ou infini) admet une borne supérieure.

Remarque (1) Si $(P, \sqsubseteq)$ est un treillis complet, alors tout ensemble $Q \subseteq P$ admet également une borne inférieure définie par $\bigsqcap Q = \bigsqcup \{x \in P \ : \ \forall y \in Q,\ x \sqsubseteq y\}$. De plus, un treillis complet admet un plus petit élément $\bot = \bigsqcup \emptyset$ et un plus grand élément $\top = \bigsqcup P$.

Remarque (2) La notion de treillis complet est plus forte que la notion de CPO. Ainsi, tout treillis complet est un CPO, mais l'inverse n'est pas vrai.

Remarque (3) Dans la suite, nous utiliserons souvent le terme de domaine pour évoquer un ensemble quelconque muni d'un ordre. Lorsque nous en aurons besoin, nous préciserons si le domaine possède une structure du CPO ou de treillis.

2.1.3 Fonctions monotones et théorèmes de point fixe

Définition 2.8 (Fonction monotone) Soit $(P, \sqsubseteq_P)$ et $(Q, \sqsubseteq_Q)$ deux CPO. On dit qu'une fonction $f : P \to Q$ est *monotone* si et seulement si :

$$\forall x, y \in P,\ x \sqsubseteq_P y \Rightarrow f(x) \sqsubseteq_Q f(y)\ . \tag{2.6}$$

Exemple 2.6 On définit la fonction valeur absolue $abs : Sign \to Sign$ par :

$$\forall x \in Sign,\ abs(x) = \begin{cases} \bot & \text{si } x = \bot \\ \top & \text{si } x = \top \\ 0 & \text{si } x = 0 \\ + & \text{si } x \in \{\varnothing, -, +\} \\ \dot{+} & \text{sinon} \end{cases} .$$

La fonction *abs* est monotone.

Définition 2.9 (Fonction continue) Soit $(P, \sqsubseteq_P)$ et $(Q, \sqsubseteq_Q)$ deux CPO. Une fonction $f : P \to Q$ est dite *continue* si et seulement si elle est monotone et si pour toute chaîne croissante X de P, on a :

$$\bigsqcup\nolimits_Q \{f(x) \ : \ x \in X\} = f\left(\bigsqcup\nolimits_P X\right)\ . \tag{2.7}$$

Une fonction continue est donc une fonction croissante qui préserve les bornes supérieures des chaînes croissantes.

Définition 2.10 (Point fixe) Soit $(P, \sqsubseteq)$ un ordre partiel, et $f : P \to P$ une fonction sur P. On dit que x est un point fixe de f si et seulement si $f(x) = x$. On dira que x est le plus petit point fixe (s'il existe) de f si et seulement si x est un point fixe de f et si $\forall y \in P,\ f(y) = y \Rightarrow x \sqsubseteq y$. Le plus petit point fixe de f, quand il existe, sera noté $lfp(f)$. De même, le plus grand point fixe de f sera noté $gfp(f)$.

Théorème 2.1 (Point fixe de fonctions continues sur un CPO) *Soit $(P, \sqsubseteq)$ un CPO, et soit $\bot$ son plus petit élément. Soit $f : P \to P$ une fonction continue sur P. Alors, f possède un plus petit point fixe $lfp(f)$ donné par :*

$$lfp(f) = \bigsqcup \left\{ f^i(\bot) \ :\ i \in \mathbb{N} \right\}. \tag{2.8}$$

Remarque Pour toute fonction $f : P \to P$, pour tout $i \in \mathbb{N}$ et tout élément $x \in P$, on définit le i-ème itéré de f sur x, noté $f^i(x)$, par : $f^0(x) = x$ et $\forall i \geq 1,\ f^i(x) = f\big(f^{i-1}(x)\big)$.

Théorème 2.2 (Théorème du point fixe de Tarsky, [Tar55]) *Soit P un treillis complet et $f : P \to P$ une fonction monotone. Alors, f admet un plus petit et un plus grand point fixe. Le plus petit point fixe sera noté $lfp(f)$ et le plus grand $gfp(f)$.*

Théorème 2.3 (Théorème du point fixe de Kleene [Bir67]) *Soit $f : P \to P$ une fonction continue sur un treillis complet P dont le plus petit (respectivement plus grand) élément est $\bot$ (respectivement $\top$). On a alors :*

$$lfp(f) = \bigsqcup \left\{ f^i(\bot) \ :\ i \in \mathbb{N} \right\} \quad gfp(f) = \bigsqcap \left\{ f^i(\top) \ :\ i \in \mathbb{N} \right\}. \tag{2.9}$$

2.2 Sémantique concrète d'un programme

La sémantique d'un programme est la description mathématique de l'ensemble des comportements possibles de ce programme. Elle permet de prédire comment le programme va s'exécuter en machine, en supposant que le compilateur respecte cette sémantique. Il existe plusieurs modèles mathématiques permettant de décrire le comportement d'un programme. La sémantique opérationnelle décrit directement l'exécution du programme via un système de transitions : on voit alors le programme comme une machine abstraite qui modifie les valeurs des variables du programme. La sémantique axiomatique décrit chaque instruction du programme comme un transformateur de propriétés logiques : un programme fera le lien entre deux propriétés logiques p et q de telle manière que si p est vraie avant l'exécution du programme, alors q est vraie après celle-ci. Les propriétés portent généralement sur les valeurs des variables du programme. La sémantique dénotationnelle traduit un programme en une fonction continue entre les états du programme : cette fonction sera définie comme la composition des fonctions élémentaires associées à chaque instruction du programme. Toutes ces sémantiques sont équivalentes, nous choisissons de présenter en détails la sémantique dénotationnelle, qui est souvent la sémantique utilisée pour définir une analyse statique par interprétation abstraite. Pour une description approfondie et une comparaison des différentes sémantiques, on pourra se référer à [Win93].

Nous commencerons donc par définir la syntaxe d'un langage impératif simple, noté **SIMPLE**, inspiré du langage **IMP** introduit dans [Win93], puis nous explicitons le domaine des états du programme et enfin nous décrivons la construction de la sémantique dénotationnelle des programmes écrits dans le langage **SIMPLE**.

2.2.1 Syntaxe du langage SIMPLE

Le langage **SIMPLE** que nous considérons est un langage impératif simple, ne comportant qu'un seul type de valeurs (les entiers) et sans pointeur. Nous supposerons disposer d'un ensemble

infini Var de noms de variables possibles. La syntaxe complète du langage est donnée par la figure 2.2 : on dispose d'expressions arithmétiques **Exp** et d'instructions **Stmt**. Une expression est soit un entier naturel $n \in \mathbb{N}$, soit une variable $X \in Var$, soit la combinaison via un opérateur binaire $\odot \in \{+, -, \times, /, =, \leq\}$ de deux expressions. Si $\odot \in \{=, \leq\}$, le résultat de l'opération sera 1 si le test est vrai, 0 sinon (voir en section 2.2.3). Les instructions sont les briques de base d'un programme. Le langage propose l'affectation $X := e$ d'une expression $e \in$ **Exp** à une variable $X \in Var$, la composition de deux instructions $s_1; s_2$, le branchement conditionnel (***if***) et un opérateur de boucle (***while***).

$$n \in \mathbb{N} \qquad X \in Var$$

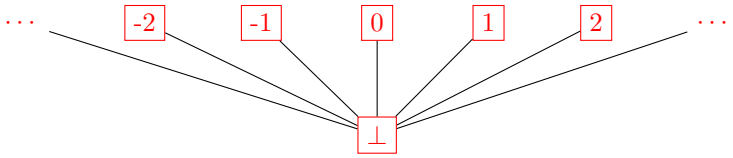
$$\begin{array}{rrcl} \textbf{Exp :} & e & ::= & n \mid X \mid e + e \mid e - e \mid e \times e \mid e/e \mid e = e \mid e \leq e \\ \textbf{Stmt :} & s & ::= & X := e \mid s; s \mid \textbf{\textit{if}}\ e\ \textbf{\textit{then}}\ c\ \textbf{\textit{else}}\ c \mid \textbf{\textit{while}}\ e\ \textbf{\textit{do}}\ c \end{array}$$

FIG. 2.2 – Syntaxe du langage **SIMPLE**.

2.2.2 États d'un programme

Pour les langages impératifs comme **SIMPLE**, l'instruction fondamentale est l'affectation qui modifie la valeur d'une variable. Les autres instructions servent essentiellement à organiser ces affectations pour qu'elles soient exécutées dans le bon ordre. La sémantique d'un programme est donc fondée sur une notion d'état qui associe à chaque variable du programme une valeur. Dans notre cas, cette valeur sera un entier naturel. Cependant, il est possible que lors de l'exécution d'un programme, certaines variables ne soient pas définies, c'est-à-dire qu'elles n'aient pas de valeur. Pour prendre en compte ce cas, nous associerons à chaque variable une valeur du domaine $\mathbb{Z}_\perp$, le CPO plat des entiers naturels.

Définition 2.11 (CPO plat des entiers naturels) Soit $\mathbb{Z}_\perp = \mathbb{Z} \cup \{\perp\}$ l'ensemble des entiers muni d'un élément spécial $\perp$. Soit $\sqsubseteq$ la relation d'ordre donnée par $\forall n \in \mathbb{Z},\ \perp \sqsubseteq n$. Le domaine plat des entiers naturels est l'ordre partiel $(\mathbb{Z}_\perp, \sqsubseteq)$, décrit graphiquement par le diagramme de Hasse ci-dessous :



Le domaine $\mathbb{Z}_\perp$ est un CPO, mais ce n'est pas un treillis car il ne possède pas de borne supérieure.

Définition 2.12 (États) Un état d'un programme est une fonction de l'ensemble des variables Var vers les éléments de $\mathbb{Z}_\perp$. Nous noterons Σ l'ensemble des états :

$$\Sigma = \left\{\sigma : Var \to \mathbb{Z}_\perp\right\}.$$

Remarque (1) Un état $\sigma \in \Sigma$ est donc une fonction totale σ des variables dans $\mathbb{Z}_\perp$, ce qui peut être vue comme une fonction partielle de Var dans $\mathbb{Z}$, dont le domaine est l'ensemble des variables telles que $\sigma(x) \neq \perp$.

Remarque (2) On munit Σ d'une structure d'ordre complet en étendant point-à-point l'ordre $\sqsubseteq$ sur $\mathbb{Z}_\bot$ aux fonctions $Var \to \mathbb{Z}_\bot$. La relation d'ordre ainsi obtenu est liée à la quantité d'information portée par un état ; on aura notamment : $\sigma \sqsubseteq \sigma'$ si et seulement si

$$\forall X \in Var,\ \sigma(X) \neq \bot \Rightarrow \sigma'(X) = \sigma(X)\ .$$

Pour un état $\sigma \in \Sigma$ et une variable $X \in Var$, nous noterons $\sigma(X) \in \mathbb{Z}_\bot$ la valeur de X dans l'état σ. Étant donné un état $\sigma \in \Sigma$, une variable $X \in Var$ et un élément $n \in \mathbb{Z}_\bot$, nous noterons $\sigma\big[X \mapsto n\big]$ l'état σ' vérifiant

$$\forall Y \in Var, \sigma'(Y) = \begin{cases} \sigma(Y) & \text{si } Y \neq X \\ n & \text{sinon} \end{cases}\ .$$

Exemple 2.7 Prenons pour exemple $Var = \{x, y, z\}$. Soit σ l'environnement associant à x la valeur 2, à y la valeur -3 et à z la valeur $\bot$, $\sigma : \begin{cases} x & \mapsto & 2 \\ y & \mapsto & -3 \\ z & \mapsto & \bot \end{cases}$. On a alors $\sigma(y) = -3$ et l'environnment $\sigma' = \sigma[z \mapsto 10][x \mapsto 4]$ est donné par $\sigma' : \begin{cases} x & \mapsto & 4 \\ y & \mapsto & -3 \\ z & \mapsto & 10 \end{cases}$.

2.2.3 Sémantique dénotationnelle

La sémantique dénotationnelle voit un programme comme une fonction partielle entre états. L'intérêt de cette vision est de pouvoir comparer facilement des programmes écrits éventuellement dans des langages différents : deux programmes seront équivalent si et seulement si ils définissent la même fonction. Un deuxième intérêt de cette approche est que la sémantique dénotationnelle est compositionnelle par nature : la sémantique d'un programme se calcule comme une composition, au sens mathématique du terme, de la sémantique de ses instructions. Nous définissons donc dans la suite la sémantique de chacune des constructions du langage.

Remarque Nous présentons les fonctions et fonctions partielles comme des ensembles de couples (x, y), c'est-à-dire comme des relation binaires. Ainsi, si on note $f = \big\{(x_1, y_1), (x_2, y_2)\big\}$, cela voudra dire que f est une fonction partielle dont le domaine est $\{x_1, x_2\}$ et telle que $f(x_1) = y_1$ et $f(x_2) = y_2$.

Pour les expressions La sémantique d'une expression $e \in \mathbf{Exp}$ est une fonction, notée $\boldsymbol{E}[\![e]\!]$, entre les états et les valeurs : $\boldsymbol{E}[\![e]\!] : \Sigma \to \mathbb{Z}_\bot$. Cette fonction est donnée par les règles de la figure 2.3. La fonction associée à une constante $n \in \mathbb{Z}$ est la fonction constante qui vaut n pour tout état $\sigma \in \Sigma$ (règle (2.10)). La fonction associée à une variable $X \in Var$ est la fonction qui à tout état $\sigma \in \Sigma$ associe la valeur de X dans l'état σ (règle (2.11)). La fonction associée à une construction arithmétique $e_0 \odot e_1$, avec $\odot \in \{+, -, \times, /\}$ est la fonction qui, pour un état σ, évalue e_0 et e_1 dans cet état puis calcule le résultat de l'opération $\odot$ (règle (2.12)). Pour ces dernières règles, les opérateurs $\odot_\bot$, pour $\odot \in \{+, -, \times, /\}$ sont des extensions simples des opérateurs arithmétiques sur les entiers au domaine $\mathbb{Z}_\bot$, c'est-à-dire que l'on a en plus $\bot \odot_\bot n = n \odot_\bot \bot = \bot$ pour tout $n \in \mathbb{Z}$ et $\odot \in \{+, -, \times, /\}$. On aura également $n/0 = \bot$ pour tout $n \in \mathbb{Z}$. La fonction associée à un test $e_0 \diamond e_1$ pour $\diamond \in \{=, \leq\}$ est la fonction qui évalue e_0 puis e_1, et renvoie 1 si le test est vérifié, et 0 sinon (règle (2.13)). On définit en effet, pour $\diamond \in \{=, \leq\}$, la fonction $\diamond_\bot : \mathbb{Z}_\bot \times \mathbb{Z}_\bot \to \mathbb{Z}_\bot$ par : $\forall x \in \mathbb{Z}_\bot,\ \bot \diamond_\bot x = x \diamond_\bot x = \bot$ et $\forall n_1, n_2 \in \mathbb{Z},\ n_1 \diamond_\bot n_2 = \begin{cases} 1 & \text{si } n_1 \diamond n_2 \\ 0 & \text{sinon} \end{cases}$.

Pour les instructions Enfin, la sémantique d'une instruction $s \in \mathbf{Stmt}$ est une fonction partielle entre les états du programme : $[\![s]\!] : \Sigma \to \Sigma$. Les règles de la figure 2.4 expliquent la construction de cette fonction. Pour l'instruction d'assignation $X := e$, la fonction associée est la fonction qui substitue dans l'état d'entrée la variable X par la valeur associée à l'expression e (règle (2.14)).

La sémantique de l'opérateur de composition $s_0; s_1$ est la fonction qui évalue d'abord la fonction associée à s_0 puis transmet le résultat à la fonction associée à s_1. Il s'agit donc de la composition des deux fonctions (règle (2.15)). La sémantique du branchement conditionnel (règle (2.16)) est la fonction qui, étant donné un état, évalue la condition dans cet état puis, en fonction du résultat, évalue la branche ***else*** (si le résultat vaut 0) ou ***then*** (dans tous les autres cas). Remarquons que si l'évaluation de l'expression renvoie $\bot$, alors le résultat de la dénotation associé au branchement conditionnel renvoie $\bot_\Sigma$, le plus petit élément du CPO Σ (c'est-à-dire la fonction associant $\bot$ à chaque variable). La sémantique d'une boucle ***while*** est exprimée comme le point fixe d'une fonction Γ définie sur le domaine des fonctions continues entre états. En effet, une instruction ***while*** e ***do*** p fonctionne ainsi : on évalue d'abord l'expression e, et si le résultat est 0, on termine. Sinon, on évalue le corps de la boucle p puis on répète l'opération. La sémantique de la boucle doit donc être définie en fonction d'elle même, d'où l'introduction de l'opérateur de point fixe dans la règle (2.17).

Remarque (1) Les fonctions définissant la sémantique des instructions sont des fonctions partielles entre états. Nous pouvons facilement étendre une fonction partielle f en une fonction totale entre états en posant $f(\sigma) = \bot_\Sigma$ si $\sigma \notin dom(f)$, où $dom(f)$ est le domaine de f et $\bot_\Sigma$ est le plus petit élément de Σ vérifiant donc $\forall X \in Var\ \bot_\Sigma(X) = \bot$. Les fonctions de la sémantique des instructions sont alors des fonctions totale continues du CPO $(\Sigma, \subseteq)$ (définition 2.12) dans lui-même.

Remarque (2) La sémantique de la boucle ***while*** implique un calcul de plus petit point fixe d'une fonction définie sur le domaine des fonctions continues de $(\Sigma, \subseteq)$ dans lui même. Comme $(\Sigma, \subseteq)$ est un CPO, le domaine des fonctions continues l'est également et donc, d'après le théorème 2.1, le plus petit point fixe existe, et on peut le calculer par une itération à la Kleene.

Exemple 2.8 Considérons le programme $P ::= \ i := 0; \textbf{\textit{while}}\ i \leq 10\ \textbf{\textit{do}}\ \big(\ X := X+1; i := i+1\big)$. Les états du programmes sont des fonctions de $\{X, i\}$ vers $\mathbb{N}^2$ et la sémantique du programme est donnée par l'équation (2.19). Cette équation signifie que l'état à la fin du programme est l'état initial dans lequel la variable i prend la valeur 10 (i vaudra toujours 10 après l'exécution de P) et la variable X est incrémentée de 10.

$$[\![P]\!] : \left\{ \begin{array}{lcl} \Sigma & \to & \Sigma \\ \sigma & \mapsto & \sigma\big[X \mapsto \sigma(X) + 10\big]\big[i \mapsto 10\big] \end{array} \right. \tag{2.19}$$

2.2.4 Sémantique collectrice

La sémantique dénotationnelle d'un programme est donc une fonction associant à un état (typiquement la valeur des variables d'entrée du programme) un autre état (typiquement la valeur des variables après l'exécution du programme). Cependant, lorsque l'on veut analyser un programme, on cherche à étudier son comportement quelles que soient les valeurs des entrées (ou du moins pour toutes les valeurs dans une certaine région). Il est donc nécessaire de définir une sémantique dite *collectrice* qui associe à un ensemble d'états (typiquement l'ensemble des entrées possibles du programme) un ensemble d'états représentant tous les résultats possibles du programme sur

$$\odot \in \{+, -, \times, /\} \qquad \diamond \in \{=, \leq\}$$

$$\begin{array}{rcll} \boldsymbol{E}[\![n]\!] & = & \big\{(\sigma, n)\ :\ \sigma \in \Sigma\big\} & (2.10) \\ \boldsymbol{E}[\![X]\!] & = & \big\{(\sigma, \sigma(X))\ :\ \sigma \in \Sigma\big\} & (2.11) \\ \boldsymbol{E}[\![e_0 \odot e_1]\!] & = & \big\{(\sigma, n_0 \odot_\bot n_1)\ :\ (\sigma, n_0) \in \boldsymbol{E}[\![e_0]\!]\ \wedge\ (\sigma, n_1) \in \boldsymbol{E}[\![e_1]\!]\big\} & (2.12) \\ \boldsymbol{E}[\![e_0 \diamond e_1]\!] & = & \big\{(\sigma, n_0 \diamond_\bot n_1)\ :\ (\sigma, n_0) \in \boldsymbol{E}[\![e_0]\!]\ \wedge\ (\sigma, n_1) \in \boldsymbol{E}[\![e_1]\!]\big\} & (2.13) \end{array}$$

FIG. 2.3 – Dénotations pour les expressions arithmétiques.

$$
\begin{array}{rcll}
[\![X := e]\!] & = & \{(\sigma, \sigma[X \mapsto n]) \ : \ \sigma \in \Sigma \wedge (\sigma, n) \in \boldsymbol{E}[\![e]\!]\} & (2.14) \\
[\![s_0; s_1]\!] & = & [\![s_1]\!] \circ [\![s_0]\!] & (2.15) \\
[\![\boldsymbol{if}\ e\ \boldsymbol{then}\ s_0\ \boldsymbol{else}\ s_1]\!] & = & \{(\sigma, \sigma') \ : \ \exists n \in \mathbb{Z}^*,\ (\sigma, n) \in \boldsymbol{E}[\![e]\!] \wedge (\sigma, \sigma') \in [\![s_0]\!]\} \cup & (2.16) \\
 & & \{(\sigma, \sigma') \ : \ (\sigma, 0) \in \boldsymbol{E}[\![e]\!] \wedge (\sigma, \sigma') \in [\![s_1]\!]\} & \\
[\![\boldsymbol{while}\ e\ \boldsymbol{do}\ s]\!] & = & lfp(\Gamma) \text{ avec} & (2.17) \\
\Gamma(\varphi) & = & \{(\sigma, \sigma') \ : \ \exists n \in \mathbb{Z}^*,\ (\sigma, n) \in \boldsymbol{E}[\![e]\!] \wedge (\sigma, \sigma') \in \varphi \circ [\![s]\!]\} \cup & \\
 & & \{(\sigma, \sigma) \ : \ (\sigma, 0) \in \boldsymbol{E}[\![e]\!]\} & (2.18)
\end{array}
$$

FIG. 2.4 – Dénotations pour les instructions.

toutes ces entrées. Nous étendons donc la sémantique dénotationnelle à des ensembles d'états de manière très classique. Un environnement ρ sera un ensemble d'états, et on notera $\mathcal{E}$ l'ensemble des environnements :

$$\mathcal{E} = \mathcal{P}(\Sigma) \ . \tag{2.20}$$

La sémantique dénotationnelle collectrice, donnée par les règles de la figure 2.5, est une extension point-à-point de la sémantique concrète aux ensembles d'états. La sémantique collectrice d'une expression $e \in \boldsymbol{Exp}$ est une fonction $\boldsymbol{E}(\!|e|\!) : \mathcal{E} \to \mathcal{P}(\mathbb{Z}_\perp)$ (règle (2.21)) et la sémantique collectrice d'une instruction $s \in \boldsymbol{Stmt}$ est une fonction $(\!|s|\!) : \mathcal{E} \to \mathcal{E}$.

$$\rho \in \mathcal{E}$$

$$
\begin{array}{rcll}
\forall e \in \mathbf{Exp},\ \boldsymbol{E}(\!|e|\!)(\rho) & = & \{n \ : \ \exists \sigma \in \rho,\ (\sigma, n) \in \boldsymbol{E}[\![e]\!]\} & (2.21) \\
\forall s \in \mathbf{Stmt},\ (\!|s|\!)(\rho) & = & \{\sigma' \ : \ \exists \sigma \in \rho,\ (\sigma, \sigma') \in [\![s]\!]\} & (2.22)
\end{array}
$$

FIG. 2.5 – Sémantique dénotationnelle étendue aux ensembles d'états.

2.3 Principe de l'interprétation abstraite

La sémantique collectrice décrit précisément le comportement réel d'un programme comme une fonction entre les environnements d'entrée et les environnements de sortie. Cette sémantique fournit toutes les informations disponibles sur le programme, elle pourrait donc servir, dans l'absolu, à prouver que le programme vérifie sa spécification. Cependant, le calcul automatique de la sémantique collectrice pose plusieurs problèmes : d'une part on doit représenter un ensemble potentiellement infini d'états et d'autre part le calcul du point fixe qui intervient dans les règles du calcul de la sémantique du ***while*** nécessite une itération potentiellement infinie de la séquence de Kleene (théorème 2.3). Le but de l'interprétation abstraite [CC77, CC92a] est de contourner ces deux problèmes en calculant non pas la sémantique concrète mais une autre sémantique, que nous appellerons sémantique abstraite, qui approxime la sémantique concrète et qui est suffisante pour prouver que le programme répond à sa spécification. Par exemple, si on veut montrer que le résultat d'un programme est toujours positif, il n'est pas nécessaire de connaître exactement l'ensemble des résultats possibles, il suffit de savoir que tous les résultats sont positifs : on peut oublier une partie de l'information (la valeur exacte du résultat) et ne garder que celle dont on a besoin (le signe). En pratique, la construction d'un analyseur statique par interprétation abstraite fonctionne ainsi : après avoir défini la sémantique concrète des programmes et les propriétés que l'on veut prouver, on choisit un domaine abstrait permettant de représenter en machine un ensemble infini d'états en ne gardant que les informations pertinentes vis-à-vis de la propriété à prouver. Ensuite,

il faut établir le lien entre le domaine abstrait et les environnements concrets définis par l'équation (2.20). Enfin, il faut définir comment chaque construction du langage modifie les éléments du domaine abstrait. On définit donc une sémantique dénotationnelle abstraite comme une fonction continue entre éléments du domaine abstrait, et on prouve la sûreté de cette sémantique abstraite vis-à-vis de la sémantique concrète. Dans la suite de la section nous expliquons en détails et avec des exemples chacune de ces étapes.

2.3.1 Domaine abstrait

Un domaine abstrait est un ensemble de valeurs symbolique $\mathcal{E}^\sharp$ représentables en machine, muni d'une fonction de concrétisation $\gamma : \mathcal{E}^\sharp \to \mathcal{E}$ qui associe à chaque élément $\rho^\sharp \in \mathcal{E}^\sharp$ un ensemble, potentiellement infini, de valeurs concrètes.

Exemple 2.9 Le domaine des signes, $Sign = \{\bot, -, +, 0, \dot{-}, \dot{+}, \varnothing, \top\}$, est un domaine abstrait, avec la fonction de concrétisation $\gamma : Sign \to \mathcal{P}(\mathbb{Z})$ définie par :

$$\gamma(\bot) = \emptyset \qquad \gamma(-) = \{n \in \mathbb{Z} \ : \ n < 0\} \qquad \gamma(+) = \{n \in \mathbb{Z} \ : \ n > 0\} \qquad \gamma(0) = \{0\}$$

$$\gamma(\dot{-}) = \{n \in \mathbb{Z} \ : \ n \leq 0\} \qquad \gamma(\dot{+}) = \{n \in \mathbb{Z} \ : \ n \geq 0\} \qquad \gamma(\varnothing) = \{n \in \mathbb{Z} \ : \ n \neq 0\} \qquad \gamma(\top) = \mathbb{Z}$$

Dans l'exemple précédent, l'élément abstrait $\dot{-}$ représente l'ensemble des entiers relatifs négatifs ou nul, alors que $-$ représente l'ensemble des entiers relatifs strictement négatifs. L'élément $-$ est donc plus précis que $\dot{-}$ en ce sens qu'il apporte plus d'information sur les valeurs concrètes qu'il représente. Pour formaliser cette notion de précision, on munit les domaines abstraits d'une structure d'ordre partiel $(\mathcal{E}^\sharp, \subseteq^\sharp)$: si $\rho_1^\sharp, \rho_2^\sharp \in \mathcal{E}^\sharp$, $\rho_1^\sharp \subseteq^\sharp \rho_2^\sharp$ signifie que $\rho_1^\sharp$ est plus précis que $\rho_2^\sharp$. Cette notion de précision sur les éléments abstraits doit cependant être liée à l'ordre sur les éléments concrets sous-jacents. En effet, le domaine concret $\mathcal{E}$ possède une structure d'ordre partiel avec l'inclusion $\subseteq$, il faut donc s'assurer que l'ordre sur les éléments abstraits soit compatible avec l'ordre sur les éléments concrets. Il faut donc que la fonction de concrétisation $\gamma : (\mathcal{E}^\sharp, \subseteq^\sharp) \to (\mathcal{E}, \subseteq)$ soit croissante :

$$\forall \rho_1^\sharp, \rho_2^\sharp \in \mathcal{E}^\sharp, \ \rho_1^\sharp \subseteq^\sharp \rho_2^\sharp \Rightarrow \gamma(\rho_1^\sharp) \subseteq \gamma(\rho_2^\sharp) \ .$$

Exemple 2.10 Le domaine abstrait $Sign$ possède une structure d'ordre partiel avec l'ordre $\sqsubseteq$ donné en section 2.1.1. On a bien par exemple $- \sqsubseteq \dot{-}$.

Les règles de la sémantique concrète montrent que l'opération la plus importante sur les environnements concrets est l'union $\cup$. Nous devons donc munir le domaine abstrait d'un opérateur d'union et d'un opérateur d'intersection abstraits $\cup^\sharp$ et $\cap^\sharp$, c'est-à-dire que nous construisons un treillis abstrait $(\mathcal{E}^\sharp, \subseteq^\sharp, \bot^\sharp, \top^\sharp, \cup^\sharp, \cap^\sharp)$. Les opérations abstraites $\cup^\sharp$ et $\cap^\sharp$ doivent être des abstractions *sûres* (et si possible efficaces) de $\cup$ et $\cap$, c'est-à-dire que l'on doit avoir :

$$\forall \rho_1^\sharp, \rho_2^\sharp \in \mathcal{E}^\sharp, \quad \gamma(\rho_1^\sharp) \cup \gamma(\rho_2^\sharp) \subseteq \gamma(\rho_1^\sharp \cup^\sharp \rho_2^\sharp) \quad \text{et} \quad \gamma(\rho_1^\sharp) \cap \gamma(\rho_2^\sharp) \subseteq \gamma(\rho_1^\sharp \cap^\sharp \rho_2^\sharp) \tag{2.23}$$

Exemple 2.11 Pour le domaine des signes, on a par exemple $- \sqcup + = \varnothing$. Comme $\gamma(-) = \{n \in \mathbb{Z} \ : \ n < 0\}$ et $\gamma(+) = \{n \in \mathbb{Z} \ : \ n > 0\}$, on a bien $\gamma(-) \cup \gamma(+) = \mathbb{Z} \backslash \{0\} = \gamma(\varnothing)$.

Dans la mesure du possible, on essaiera également d'avoir une fonction d'abstraction $\alpha : \mathcal{E} \to \mathcal{E}^\sharp$, croissante, qui associe à chaque ensemble de valeurs concrètes une valeur abstraite. Les fonctions d'abstraction et de concrétisation doivent être compatibles avec les ordres concrets et abstraits, c'est-à-dire que (α, γ) doit former une correspondance de Galois.

Définition 2.13 (Correspondance de Galois) Soit $(E, \sqsubseteq_E)$ et $(F, \sqsubseteq_F)$ deux ordres partiels. Une correspondance de Galois entre E et F consiste en deux fonctions monotones $\alpha : E \to F$ et $\gamma : F \to E$ telles que :

$$\forall x \in E, \forall y \in F, \quad \alpha(x) \sqsubseteq_F y \iff x \sqsubseteq_E \gamma(y) \ . \tag{2.24}$$

Exemple 2.12 La fonction $\alpha : \mathcal{P}(\mathbb{Z}) \to Sign$ définie par :

$$\forall X \in \mathcal{P}(\mathbb{Z}),\ \alpha(X) = \begin{cases} \bot & \text{si } X = \emptyset \\ - & \text{si } \forall n \in X,\ n < 0 \\ + & \text{si } \forall n \in X,\ n > 0 \\ \dot{-} & \text{si } \forall n \in X,\ n \leq 0 \text{ et } 0 \in X \\ \dot{+} & \text{si } \forall n \in X,\ n \geq 0 \text{ et } 0 \in X \\ 0 & \text{si } X = \{0\} \\ \varnothing & \text{si } 0 \notin X \\ \top & \text{sinon} \end{cases}$$

est une fonction d'abstraction et (α, γ) est une correspondance de Galois.

Une fonction d'abstraction telle que (α, γ) soit une correspondance de Galois n'existe pas pour tous les domaines abstraits (par exemple pour les polyèdres). Quand elle existe, on dira que $\alpha(\rho)$ est la meilleure abstraction de $\rho \in \mathcal{E}$, et la meilleure abstraction de l'opérateur d'union (respectivement d'intersection) sera alors définie par $\alpha \circ \cup \circ \gamma$ (respectivement $\alpha \circ \cap \circ \gamma$). Ces meilleures abstractions peuvent cependant être coûteuses à calculer, on ne les utilisera donc pas toujours, même lorsqu'elles existent.

Le choix d'un bon domaine abstrait est très important lors de l'analyse d'un programme. Il existe en effet de nombreux domaines qui apportent chacun plus ou moins d'information selon leur niveau d'abstraction. Outre le domaine des signes déjà présenté, citons le domaine des intervalles $\mathcal{E}_I^\sharp = Var \to \mathbb{I}^\sharp$, où $\mathbb{I}^\sharp$ est l'ensemble des intervalles défini par l'équation (2.25).

$$\mathbb{I}^\sharp = \{[a, b]\ :\ a, b \in \mathbb{Z} \cup \{-\infty\} \cup \{+\infty\}\} \cup \{\bot_I^\sharp\} \tag{2.25}$$

On muni aisément $\mathbb{I}^\sharp$ d'une structure de treillis et on étend cette structure à $\mathcal{E}_I^\sharp$ point-à-point. Le problème du domaine des intervalles est qu'il oublie les relations entre les variables du programme. Par exemple, la valeur concrète $\rho = \left\{\{x \mapsto i; y \mapsto i\}\ :\ i \in [-10, 10]\right\}$ sera abstraite par $\rho^\sharp = \{x \mapsto [-10, 10]\,; y \mapsto [-10, 10]\}$. On a donc perdu une information essentielle : pour tout élément $\sigma \in \rho$, on a $\sigma(x) = \sigma(y)$; en effet, la concrétisation de $\rho^\sharp$ est $\left\{\{x \mapsto i; y \mapsto j\}\ :\ i, j \in [-10, 10]\right\}$, qui ne garantit plus cette propriété.

Pour répondre à ce problème, des domaines dits *relationnels* ont été définis. Ainsi, le domaine des polyèdres [CH78] représente les éléments abstraits par un polyèdres convexe clos, c'est-à-dire un ensemble d'inégalités linéaires de la forme $\sum_i \alpha_i X_i \leq \beta_i$ où $X_i \in Var,\ \alpha_i, \beta_i \in \mathbb{N}$. On garde donc en mémoire des relations linéaires entre les variables du programme, ce qui permet une abstraction nettement plus précise des ensembles de valeurs concrètes. Cependant, la complexité algorithmique des calculs sur le domaine des polyèdres est souvent trop élevée pour des programmes de très grande taille. Des domaines relationnels plus simples ont donc été introduits dans le but de garder certaines relations entre les variables tout en ayant une représentation et des calculs efficaces. Citons par exemple le domaine des octogones [Min01] qui limite les inégalités linéaires à des inégalités de la forme $\pm X_i \pm X_j \leq c$ avec $X_i, X_j \in Var$ et $c \in \mathbb{N}$. On ne garde donc en mémoire que certaines relations entre les variables. Le résultat est donc une représentation moins précise des ensembles de valeurs concrets, mais les calculs sont nettement plus efficaces. Une comparaison graphique de la représentation d'un ensemble de valeurs concrètes (points verts) par un intervalle (en bleu), un octogone (en rouge) et un polyèdres (en gris) est donnée à la figure 2.6.

Les domaines que nous venons de mentionner sont utilisés pour calculer l'ensemble de valeurs prises par les variables d'un programme au cours de toutes ses exécutions. Comme nous l'avons mentionné dans l'introduction, il est souvent également intéressant de prouver d'autres propriétés sur les logiciels embarqués, comme par exemple des propriétés de précision numériques [GMP02]. Pour cette propriété, des domaines qui ne donnent que la plage de valeurs prises par les variables ne sont pas suffisant, on a besoin d'information supplémentaire quant à l'erreur de calcul commise à cause de l'utilisation de nombres à virgule flottante. Pour cela, le domaine de l'erreur globale, qui représente un nombre réel comme un nombre flottant plus une erreur de calcul, ou le domaine des séries d'erreurs [Gou01, Mar02, Mar06], qui décompose l'erreur en fonction des instructions du

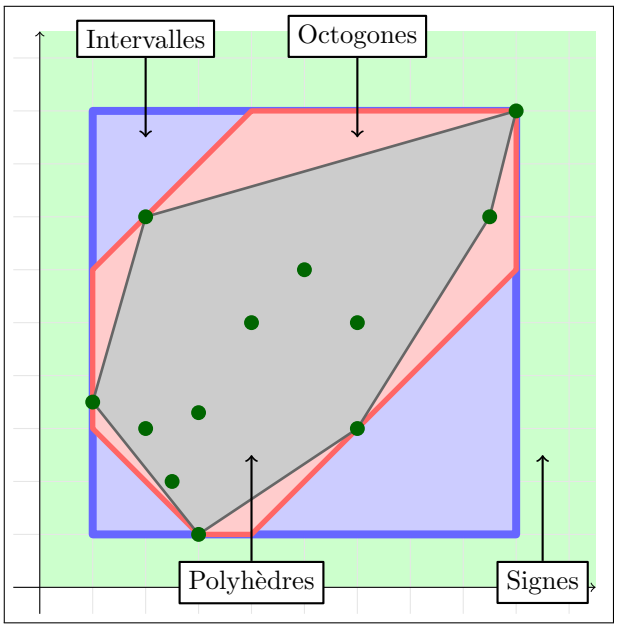


FIG. 2.6 – Différentes abstractions d'un même ensemble de points.

programme, ont été utilisé avec succès. Un comparatif des domaines abstraits pour la vérification des propriétés numériques des programmes est disponible dans [Mar05].

2.3.2 Sémantique abstraite

Pour calculer la sémantique abstraite, nous définissons pour chaque instruction du langage $s \in \mathbf{Stmt}$ une fonction abstraite $(\!|s|\!)^\sharp$ agissant sur les environnements abstraits. Cette fonction doit surapproximer la sémantique concrète $(\!|s|\!)$, c'est-à-dire que nous devons définir $(\!|s|\!)^\sharp$ de telle sorte que :

$$\forall \rho^\sharp \in \mathcal{E}^\sharp,\ \gamma\big((\!|s|\!)^\sharp(\rho^\sharp)\big) \subseteq (\!|s|\!)\big(\gamma(\rho^\sharp)\big) \tag{2.26}$$

Autrement dit, le calcul de la sémantique abstraite doit être moins précis que celui de la sémantique concrète : si on effectue d'abord la fonction abstraite puis la concrétisation, on obtient un ensemble plus grand que si on effectue d'abord la concrétisation puis le calcul de la sémantique concrète (voir figure 2.7, les rectangles bleus sont les éléments concrets, les ensembles rouges leurs concrétisations et l'ensemble gris le résultat de la sémantique concrète).

La sémantique abstraite associée à une affectation $X := e$ dépend fortement du domaine numérique choisit pour représenter les ensembles de valeurs. Par exemple, dans le domaine des intervalles l'affectation $Y := X + 3$ changera dans l'environnement abstrait la valeur de la variable Y en le résultat de l'opération $X + 3$ calculée par une arithmétique d'intervalle. Dans le domaine des octogones, la même instruction revient à ajouter les deux contraintes $Y - X \leq 3$ et $X - Y \leq -3$. Pour l'opérateur de composition, la règle sémantique est la même que pour le cas concret : $(\!|s_1; s_2|\!)^\sharp = (\!|s_1|\!)^\sharp \circ (\!|s_2|\!)^\sharp$.
On peut définir la sémantique abstraite du branchement conditionnel ***if*** ainsi :

$$(\!|\textbf{\textit{if}}\ e\ \textbf{\textit{then}}\ s_1\ \textbf{\textit{else}}\ s_2|\!)^\sharp(\rho^\sharp) = (\!|s_1|\!)^\sharp(\rho) \cup^\sharp (\!|s_2|\!)^\sharp(\rho)\ .$$

On calcule donc la sémantique du ***if*** en calculant l'effet des deux branches ***then*** et ***else*** sur l'état abstrait puis en effectuant l'union des deux résultats. Cette définition vérifie les critères de sûreté (équation (2.26)) mais effectue une grande surapproximation : on oublie complètement l'effet filtrant de l'expression e. Une meilleure solution consiste donc à définir pour chaque expression $e \in \mathbf{Exp}$ un opérateur abstrait $(\!|e|\!)^\sharp : \mathcal{E}^\sharp \to \mathcal{E}^\sharp$ qui surapproxime l'effet du test ***if*** e. Pour un

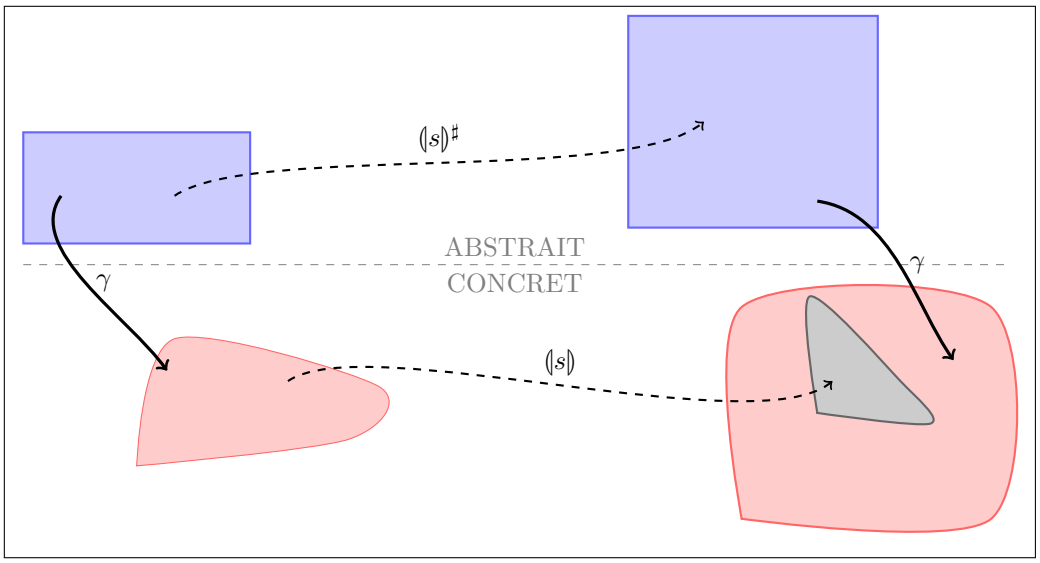


FIG. 2.7 – Critère de sûreté pour la sémantique abstraite.

environnement abstrait $\rho^\sharp \in \mathcal{E}^\sharp$, $(\!|e|\!)^\sharp(\rho)$ est une surapproximation de l'ensemble des états concrets vérifiant e :

$$\forall \rho^\sharp \in \mathcal{E}^\sharp,\ \big\{\sigma \in \gamma(\rho^\sharp)\ :\ \boldsymbol{E}[\![e]\!](\sigma) \neq 0\big\} \subseteq \gamma\big((\!|e|\!)^\sharp(\rho^\sharp)\big)\ .$$

On définit également pour tout $e \in \mathbf{Exp}$ un opérateur $\neg(\!|e|\!)^\sharp : \mathcal{E}^\sharp \to \mathcal{E}^\sharp$ vérifiant :

$$\forall \rho^\sharp \in \mathcal{E}^\sharp,\ \big\{\sigma \in \gamma(\rho^\sharp)\ :\ \boldsymbol{E}[\![e]\!](\sigma) = 0\big\} \subseteq \gamma\big(\neg(\!|e|\!)^\sharp(\rho^\sharp)\big)\ .$$

En utilisant $(\!|e|\!)^\sharp$, on peut alors définir une sémantique abstraite plus précise pour le branchement conditionnel :

$$(\!|\boldsymbol{if}\ e\ \boldsymbol{then}\ s_0\ \boldsymbol{else}\ s_1|\!)^\sharp(\rho^\sharp) = (\!|s_0|\!)^\sharp \circ (\!|e|\!)^\sharp(\rho^\sharp) \cup^\sharp (\!|s_1|\!)^\sharp \circ \neg(\!|e|\!)^\sharp(\rho^\sharp)$$

La même logique intervient pour la sémantique abstraite de la boucle ***while*** :

$$\begin{array}{rcl} (\!|\boldsymbol{while}\ e\ \boldsymbol{do}\ s|\!)^\sharp(\rho^\sharp) & = & \neg(\!|e|\!)^\sharp\left(lfp(\Gamma^\sharp)\right) \text{ avec} \\ \Gamma^\sharp(\rho') & = & \rho^\sharp \cup^\sharp \left((\!|s|\!)^\sharp \circ (\!|e|\!)^\sharp\right)(\rho') \end{array}$$

Exemple 2.13 Dans le domaine des intervalles, on a par exemple :

$$\forall c \in \mathbb{Z},\ \forall x \in Var,\ \left\{\begin{array}{rcl} (\!|x \leq c|\!)^\sharp(\rho^\sharp) & = & \rho^\sharp\big[x \mapsto \rho^\sharp(x) \cap]-\infty, c]\ \big] \\ \neg(\!|x \leq c|\!)^\sharp(\rho^\sharp) & = & \rho^\sharp\big[x \mapsto \rho^\sharp(x) \cap [c+1, \infty[\ \big] \end{array}\right.$$

Le calcul de la sémantique abstraite nécessite donc le calcul du plus petit point fixe d'une fonction $\Gamma^\sharp$ entre environnements abstraits. Ce plus petit point fixe existe d'après le théorème 2.2 car le domaine abstrait est un treillis complet et $\Gamma^\sharp$ est une fonction continue. On peut donc calculer le plus petit point fixe par une itération de Kleene. Le critère de sûreté de l'équation (2.26) s'étend par composition et passage au point fixe, de sorte que pour tout programme P et tout environnement abstrait $\rho \in \mathcal{E}^\sharp$, on a :

$$(\!|P|\!)\big(\gamma(\rho^\sharp)\big) \subseteq \gamma\big((\!|P|\!)^\sharp(\rho^\sharp)\big)\ . \tag{2.27}$$

Exemple 2.14 Reprenons le programme $P ::= \ i := 1; X := 0; \boldsymbol{while}\ i \leq 10\ \boldsymbol{do}\big(\ X := X+1; i := i+1\big)$ de l'exemple 2.8. La sémantique abstraite de P en utilisant le domaine abstrait $Sign$ est :

$$(\!|P|\!)^\sharp : \left\{\begin{array}{rcl} \mathcal{E}^\sharp & \to & \mathcal{E}^\sharp \\ \rho^\sharp & \mapsto & \big\{i \mapsto +, X \mapsto \dot{+}\big\} \end{array}\right.$$

On montre donc que pour toute exécution du programme, le résultat (à savoir les valeurs de X et i) est positif. On montre même que la valeur de i est strictement positive. En revanche, on ne peut pas montrer que X est différent de 0, car la sémantique de l'expression booléenne $i \leq 10$ dans le domaine des signes est la fonction identité. On ne peut donc pas, après le calcul du point fixe de la sémantique du ***while***, réduire l'élément abstrait obtenu grâce à $\neg(\!|i \leq 10|\!)^\sharp$.

2.3.3 Accélération de la convergence

L'utilisation d'un domaine abstrait et de la sémantique abstraite permettent de résoudre le premier problème du calcul de la sémantique collectrice, à savoir manipuler des ensembles potentiellement infini de valeurs. Cependant, le deuxième problème (le calcul d'un point fixe par une itération de Kleene potentiellement infinie) n'est pas résolu. En effet, si le domaine abstrait choisi a une hauteur infinie, le calcul de la sémantique de la boucle ***while*** peut demander une itération infinie. Même si le treillis est de hauteur finie, le nombre d'itérations nécessaires pour atteindre le point fixe peut être rédhibitoire pour l'analyse de programmes de grande taille. La théorie de l'interprétation abstraite repose donc sur une deuxième technique, appelée élargissement (*widening* en anglais) [CC77, CC92b], qui assure la terminaison du calcul de la sémantique au prix d'une approximation supplémentaire.

L'intuition derrière le widening est la suivante : lorsqu'on calcule le point fixe de la fonction $\Gamma^\sharp$ par une itération de Kleene, on calcule une chaîne croissante d'éléments $\rho_0^\sharp \subseteq^\sharp \rho_1^\sharp \subseteq^\sharp \ldots \subseteq^\sharp \rho_n^\sharp \subseteq^\sharp \ldots$ définis par $\rho_0^\sharp = \bot^\sharp$ et $\rho_{n+1}^\sharp = \Gamma^\sharp(\rho_n)$. Cette suite converge vers le point fixe $lfp(\Gamma^\sharp)$. Plutôt que de calculer $lfp(\Gamma^\sharp)$, la technique du widening propose de transformer cette chaîne croissante en une suite stationnaire qui converge vers une surapproximation du point fixe, ce qui permet de garantir la condition de sûreté de l'équation (2.27) et la terminaison des calculs. Formellement, un opérateur de widening $\triangledown$ sur un treillis E est un opérateur $\triangledown : E \times E \to E$ tel que :

1. $\forall x, y \in E,\ x \sqsubseteq x \triangledown y$
2. $\forall x, y \in E,\ y \sqsubseteq x \triangledown y$
3. pour toute suite $\left(x_n\right)_{n\in\mathbb{N}} \in E^{\mathbb{N}}$, la suite $\left(y_n\right)_{n\in\mathbb{N}}$ définie par $y_0 = x_0$ et pour tout entier $n \in \mathbb{N},\ y_{n+1} = y_n \triangledown x_{n+1}$ n'est pas strictement croissante. Remarquons que cela impose qu'elle soit ultimement stationnaire car elle est croissante d'après les propriétés 1 et 2 de l'opérateur $\triangledown$.

Exemple 2.15 Sur le treillis des intervalles $\mathbb{I}^\sharp$, on peut définir un opérateur de widening $\triangledown$ par :

$$[a,b] \triangledown [a',b'] = [c,d] \text{ avec } c = \begin{cases} -\infty & \text{si } a' < a \\ a & \text{sinon} \end{cases} \text{ et } d = \begin{cases} +\infty & \text{si } b' > b \\ b & \text{sinon} \end{cases}.$$

En pratique, on utilisera l'opérateur de widening pour calculer, au lieu de la suite $(\rho_n^\sharp)_{n\in\mathbb{N}}$ des itérés de Kleene, la suite $\left(y_n\right)_{n\in\mathbb{N}}$ définie par $y_0 = \rho_0^\sharp$ et $y_{n+1} = y_n \triangledown \rho_{n+1}^\sharp$ si $n \geq n_0$ et $y_{n+1} = y_n$ sinon. Le seuil n_0 permet d'améliorer la précision du calcul en ne déclenchant le widening qu'après un certain nombre d'itérations. On arrête ensuite les itérations lorsque l'on trouve y_l tel que $y_l = y_{l+1}$. On sait alors que $y_l \supseteq^\sharp lfp(\Gamma^\sharp)$ et on choisit y_l comme surapproximation du plus petit point fixe de $\Gamma^\sharp$. L'utilisation du widening assure donc la terminaison des calculs, mais ne garantit plus le calcul du plus petit point fixe.

Pour améliorer ensuite la précision du calcul, on pourra utiliser un opérateur de rétrécissement (*narrowing*) qui permet de raffiner le résultat obtenu. Un opérateur de narrowing sur un treillis E est une fonction $\triangledown : E \times E \to E$ telle que :

1. $\forall x, y \in E,\ x \sqcap y \sqsubseteq x \vartriangle y$;
2. $\forall x, y \in E,\ x \vartriangle y \sqsubseteq x$;
3. pour toute suite $\left(x_n\right)_{n\in\mathbb{N}} \in E^{\mathbb{N}}$, la suite décroissante $\left(y_n\right)_{n\in\mathbb{N}}$ définie par $y_0 = x_0$ et pour tout entier $n \in \mathbb{N},\ y_{n+1} = y_n \vartriangle x_{n+1}$ est ultimement stationnaire.

Le narrowing est donc une surapproximation de l'intersection permettant de réduire la surapproximation due au widening. Nous ne détaillerons pas plus la notion de narrowing car nous ne l'utiliserons pas dans la suite. On pourra se référer à [CC92b] pour plus de détails.

Exemple 2.16 Sur le treillis des intervalles $\mathbb{I}^\sharp$, on peut définir un opérateur de narrowing $\triangle$ par :

$$[a,b] \triangle [a',b'] = [c,d] \text{ avec } c = \begin{cases} a' & \text{si } a = -\infty \\ a & \text{sinon} \end{cases} \quad \text{et } d = \begin{cases} b' & \text{si } b = \infty \\ b & \text{sinon} \end{cases} .$$

2.3.4 Outils

Il existe de nombreux outils d'analyse statique par interprétation abstraite, souvent sous la forme de prototypes académiques. Peu ont cependant été utilisés pour des projets industriels de grande taille. Nous en décrivons ici trois qui sont chacun spécialisé dans un type de propriété à prouver. Mentionnons également d'autres outils : Polyspace est un outil commercial maintenant intégré à Matlab/Simulink qui prouve l'absence d'erreurs à l'exécution ; CGS (C Global Surveyor, [VB04]) est un prototype développe à la NASA qui vérifie le même type de propriétés ; TVLA [BLARS07] est un outil développé par l'université de Tel-Aviv qui vérifie des propriétés de cohérence de la mémoire (essentiellement du tas).

ASTREE

Le logiciel ASTREE[1] [BCC+03, CCF+05] (pour **A**nalyseur **s**tatique de logiciels **t**emps-**r**éel **e**mbarqués) est un analyseur statique, développé à l'École Normale Supérieure, qui prouve l'absence d'erreur à l'exécution (RTE) pour des programmes embarqués temps-réel écrits en C. ASTREE peut détecter des erreurs spécifiques au langage C (division par zéro, dépassement de capacité pour un tableau), des erreurs liées à la représentation en machine des données (dépassement de la plus grande valeur représentable par exemple) ou encore le non-respect de propriétés définies par l'utilisateur via des assertions. ASTREE ne vise pas à être un analyseur générique mais se spécialise au contraire pour une classe précise de programmes pour lesquels une RTE est critique. De par cette spécialisation et grâce à un choix de domaines abstraits spécifiques à l'analyse des logiciels visés (par exemple [Fer04] pour les filtres intervenant dans les logiciels de contrôle-commande), ASTREE atteint une très grande précision d'analyse, ce qui permet son application pour des projets industriels de très grande taille [DS07].

AbsInt

Le logiciel AbsInt[2] [FHL+01] calcule une borne supérieure sur le pire temps d'exécution (*worst wase execution time*, ou WCET) d'un programme temps-réel. AbsInt utilise un programme compilé, exécutable et reconstruit le graphe de flot de contrôle, éventuellement avec l'aide d'informations fournies par l'utilisateur relatives au nombres d'appels récursifs par exemple. L'outil effectue alors plusieurs analyses successives pour estimer le WCET : une analyse de valeurs pour déterminer le nombre maximal de tours de boucles, puis une analyse de cache pour déterminer quels accès mémoires utilisent la mémoire cache uniquement, et enfin une analyse de pipeline pour estimer le temps d'exécution de chaque instruction. Ces différentes analyses sont alors combinées pour estimer le WCET. Comme ASTREE, AbsInt a été utilisé notamment par Airbus lors du développement du programme de vol de l'A380.

Fluctuat

Le logiciel Fluctuat [3] [GMP02] mesure l'erreur due à l'utilisation des nombres flottants au lieu des nombres réels lors de l'exécution d'un programme écrit en C. L'erreur est même décomposé en une "série d'erreurs" qui indique la contribution de chaque instruction du programme à l'erreur finale. On peut ainsi visualiser directement quelle instruction cause la plus grande erreur ce qui permet un débuggage précis du programme en cas d'erreur finale trop importante. L'utilisation de domaines très spécialisés pour l'analyse des erreurs de calculs [GP06] et un grand choix de

[1] `http ://www.astree.ens.fr/`
[2] `http ://www.absint.com`
[3] `http ://www-list.cea.fr/labos/fr/LSL/fluctuat/index.html`

paramètres concernant l'analyse permettent d'obtenir des résultats précis pour une grande gamme de codes numériques [GMP06].

CHAPITRE 3

Analyse de la partie continue : intégration garantie

L'analyse de la partie continue des systèmes hybrides relève d'un domaine de compétences très différent de l'analyse statique des programmes discrets. En effet, l'évolution continue du système est généralement décrite par une (ou plusieurs) équation différentielle, et l'étude de ces objets mathématiques est un domaine extrêmement large. Comme la résolution formelle d'une équation différentielle est un problème très difficile dans le cas général, de nombreux travaux portent sur leur résolution *numérique*, c'est-à-dire trouver une suite de valeurs qui approchent au mieux la solution. De nombreuses méthodes ont été proposées pour cela (Euler, Runge-Kutta, Heun, etc.), chacune ayant ses avantages. Cependant, lorsqu'on s'intéresse à la vérification de systèmes critiques, une approximation de la solution ne suffit pas, il est nécessaire d'avoir des garanties sur l'emplacement de la solution. Ce chapitre montre les méthodes classiques permettant cela. Nous commencerons par rappeler les bases de la théorie des équations différentielles (section 3.1), puis nous détaillerons la méthode d'intégration garantie à base de séries de Taylor (section 3.3). Enfin, nous présenterons les outils existants répondant au problème de l'intégration garantie (section 3.4). Outre la section 3.1 qui se fonde sur les livres [Inc56, SB93], les travaux présentées dans ce chapitre proviennent essentiellement de [NJC99, Ned06] et [Sta97].

3.1 Rappels sur les équations différentielles

Dans cette section, nous rappelons les définitions de base ainsi que les principaux théorèmes de la théorie des équations différentielles qui sont nécessaires à la bonne compréhension des techniques d'intégration garantie. On pourra trouver plus de détails ainsi que les preuves des résultats présentés ici dans de nombreux livres sur les équations différentielles, par exemple [Inc56, SB93]. Dans toute cette section, $F : \mathbb{R}^n \to \mathbb{R}^n$ désigne une fonction continue.

Définition 3.1 (Équation différentielle) Une équation différentielle ordinaire (EDO) de dimension n est une relation entre une fonction $y : \mathcal{D} \to \mathbb{R}^n$ définie sur un ouvert $\mathcal{D}$ et sa dérivée temporelle $\dot{y} : \mathcal{D} \to \mathbb{R}^n$, donnée par :

$$\dot{y}(t) = F(y(t), t) \ . \tag{3.1}$$

Par convention, nous noterons souvent l'EDO $\dot{y} = F(y, t)$.

On peut de même définir les équations différentielles d'ordre $p > 1$ comme une relation entre y et ses p premières dérivées temporelles. Cependant, on peut toujours se ramener au cas des EDO d'ordre 1 en introduisant des dimensions supplémentaires correspondant aux dérivées de y.

Définition 3.2 (Équation différentielle autonome) Une équation différentielle est dite autonome si la fonction F ne dépend pas du temps t, c'est-à-dire que l'on a $\dot{y} = F(y)$.

Remarquons que toute équation différentielle est équivalente à une équation différentielle autonome : il suffit d'introduire une nouvelle dimension correspondant au temps, et dont la dérivée est constante égale à 1. Dans la suite, nous ne considérerons donc plus que des équations différentielles autonomes de dimension n, que nous nommerons équations différentielles de dimension n par abus de langage.

Définition 3.3 (Solution d'une équation différentielle) La solution d'une équation différentielle $\dot{y} = F(y)$ de dimension n sur un ouvert $\mathcal{D} \subseteq \mathbb{R}$ est une fonction continûment dérivable $\Phi : \mathcal{D} \to \mathbb{R}^n$ telle que

$$\forall t \in \mathcal{D},\ \dot{\Phi}(t) = F\big(\Phi(t)\big)\ . \tag{3.2}$$

Exemple 3.1 L'équation différentielle d'ordre 2

$$\left\{ \begin{array}{lcl} \dot{y_1} & = & y_2 \\ \dot{y_2} & = & -y_1 \end{array} \right.$$

admet pour solution l'ensemble des fonctions $\Phi(t) = \big(\phi_1(t), \phi_2(t)\big)$ telles que

$$\exists \alpha, \beta \in \mathbb{R}\ :\ \left\{ \begin{array}{lcl} \phi_1(t) & = & \alpha \sin(t) + \beta \cos(t) \\ \phi_2(t) & = & \alpha \cos(t) - \beta \sin(t) \end{array} \right. .$$

En général, s'il existe une solution à l'équation différentielle sur un domaine donné, alors il en existe une infinité qui sont obtenues par transformation à partir de la première solution. Par contre, si une *condition initiale* est fixée, les solutions sont (dans les bons cas) uniques. Cela nous amène donc à définir la notion de *Problème de Cauchy* (*Initial Value Problem* ou IVP en anglais).

Définition 3.4 (Problème de Cauchy) Étant donnée une équation différentielle $\dot{y} = F(y)$ et une condition initiale $y(0) = y_0$, le problème de Cauchy consiste à trouver un ouvert $D \subseteq \mathbb{R}$ qui contienne 0 et une solution Φ à l'équation différentielle sur D telle que $\Phi(0) = y_0$

Remarquons que dans le cas des EDO autonomes, le choix de 0 comme temps initial n'a pas d'importance : si $\Phi(t)$ est une solution au problème de Cauchy pour une condition initiale $y(0) = y_0$, alors $\Phi(t + t_0)$ est une solution au problème de Cauchy pour la condition initial $y(t_0) = y_0$. Nous ne considérerons donc dès maintenant que des problèmes de Cauchy pour lesquels la condition initiale est donnée en 0.

Remarque Dans la suite, nous utiliserons indifféremment le terme *problème de Cauchy* ou *IVP*.

Exemple 3.2 L'unique solution au problème de Cauchy donné par l'équation différentielle de l'exemple 3.1 et par la condition initiale $y_1(0) = 0.6,\ y_2(0) = 1.5$ est $\Phi(t) = \big(\phi_1(t), \phi_2(t)\big)$ avec :

$$\left\{ \begin{array}{lcl} \phi_1(t) & = & 3/2 \sin(t) + 3/5 \cos(t) \\ \phi_2(t) & = & 3/2 \cos(t) - 3/5 \sin(t) \end{array} \right. .$$

Selon les propriétés de la fonction F, un problème de Cauchy peut avoir une, plusieurs ou une infinité de solutions, ou alors peut ne pas en avoir du tout. Lorsqu'un IVP possède une solution, on cherche le plus souvent à trouver la solution maximale, c'est-à-dire celle qui est définie sur le plus grand ouvert. Formellement, nous dirons qu'un solution Φ_1 définie sur D_1 est une extension de Φ_2 définie sur D_2 si $D_2 \subset D_1$ et si il existe $t \in D_2$ tel que $\Phi_1(t) = \Phi_2(t)$ (les deux fonctions sont alors égales sur tout l'ouvert D_2). Une solution maximale est une solution qui ne peut pas être étendue. De nombreux résultats existent sur l'existence et l'unicité de solutions maximales pour différentes classes d'équations différentielles. Le plus utile et le plus connu concerne les équations différentielles pour lesquelles la fonction F est (au moins localement) Lipschitz.

Définition 3.5 (Fonction *k*-lipschitz) Une fonction $\Psi : \mathcal{D} \to \mathcal{E}$ définie sur un ouvert $\mathcal{D}$ muni de la distance $d_{\mathcal{D}}$ à valeur dans le domaine $\mathcal{E}$ muni de la distance $d_{\mathcal{E}}$ est dite k-lipschitz si et seulement si :

$$\forall x, y \in \mathcal{D},\ d_{\mathcal{E}}\big(\Psi(x), \Psi(y)\big) \leq k \times d_{\mathcal{D}}\big(x, y\big)\ .$$

Autrement dit, une fonction est k-lipschitz si son taux d'accroissement est borné. En particulier, une fonction k-lipschitz avec $k \leq 1$ est dite contractante. Pour les équations différentielles définies par une fonction Lipschitz, on dispose du théorème de Cauchy-Lipschitz qui assure l'existence et l'unicité d'une solution maximale, et donc nous donnons ici (théorème 3.1) une version simplifiée.

Théorème 3.1 (Existence et unicité de la solution maximale.) *Si la fonction F est k-lipschitz autour de la condition initial y_0, alors le problème de Cauchy $\dot{y} = F(y),\ y(0) = y_0$ admet une unique solution maximale définie sur un voisinage de 0.*

Ce théorème permet donc de prouver l'existence d'une unique solution à un problème de Cauchy. Cependant, il ne donne aucune caractérisation de cette solution ni aucun moyen de la calculer. Le théorème suivant, dû à Picard, donne une caractérisation de cette solution via un théorème de point fixe.

Théorème 3.2 (Théorème de Picard.) *Soit $I = \big[\underline{I}, \overline{I}\big]$ un compact de $\mathbb{R}$ contenant 0. On définit l'opérateur de Picard par*

$$P_I\big(F, y_0\big) : \left\{ \begin{array}{l} \mathcal{C}^0\big(I\big) \to \mathcal{C}^0\big(I\big) \\ f \mapsto \lambda x.y_0 + \int_{\underline{I}}^{x} F\big(f(s)\big)ds \end{array} \right. . \tag{3.3}$$

On a alors : le problème de Cauchy $\dot{y} = F(y),\ y(0) = y_0$ admet une solution sur I si et seulement si $P_I\big(F, y_0\big)$ admet un point fixe. Ce point fixe est la solution au problème de Cauchy.

Le théorème 3.2 donne donc une caractérisation de la solution à un problème de Cauchy comme un point fixe, au sens de la théorie des points fixes de Banach, d'un opérateur agissant sur les fonctions continues. Dans le cas où la fonction F est globalement lipschitz sur $\mathbb{R}^n$, on dispose même d'un théorème plus fort qui donne un moyen de calculer la solution.

Théorème 3.3 (Convergence des itérés de Picard.) *Si F est globalement lipschitz sur $\mathbb{R}^n$, la suite de fonctions définie par $f_0 \in \mathcal{C}^0\big(I\big)$ et $f_{n+1} = P_I\big(F, y_0\big)(f_n)$ est uniformément convergente pour tout compact I contenant 0. Clairement, la limite de la suite est le point fixe de $P_I\big(F, y_0\big)$ et donc la solution au problème de Cauchy.*

Le théorème 3.3 donne donc, théoriquement, un moyen de calculer la solution d'une équation différentielle, sous réserve que la fonction F soit suffisamment régulière. Cependant, le calcul de $P_I\big(F, y_0\big)$ est en pratique impossible pour des fonctions F non triviales.

Nous énonçons maintenant une des propositions fondamentales des solutions à un problème de Cauchy dont la fonction F est globalement Lipschitz. Cette proposition permet d'encadrer (de manière très large) la solution du problème de Cauchy sur tout son domaine de définition.

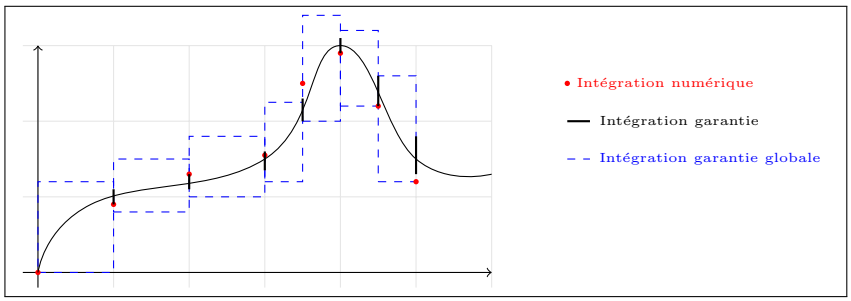


FIG. 3.1 – Comparaison de l'intégration numérique et garantie.

Proposition 3.4 *Soit $\dot{y} = F(y),\ y(t_0) = y_0$ un problème de Cauchy tel que F est α-Lipschitz. Soit y_∞ la solution unique au problème de Cauchy. On a alors :*

$$\forall t \geq t_0,\ |y(t) - y(t_0)| \leq F(y_0) \cdot e^{\alpha(t-t_0)}(t - t_0)\ . \tag{3.4}$$

3.2 But de l'intégration garantie

À partir de maintenant, le terme *problème de Cauchy* se référera au problème de Cauchy $\dot{y} = F(y),\ y(0) = y_0$ où la fonction F est globalement Lipschitz sur $\mathbb{R}$ (ou du moins globalement Lipschitz sur le codomaine de y). Nous noterons désormais y_∞ la solution du problème de Cauchy. Cette solution est généralement inconnue mais on sait en calculer des approximations grâce à des schémas d'intégration numérique. Le but de l'intégration garantie est tout autre : il s'agit de trouver des encadrements sûrs de la solution, c'est-à-dire des formes géométriques (en général des rectangles) dont on est sûr qu'elles contiennent la vraie solution, au moins à un certain nombre de points dans le temps. La figure 3.1 montre le principe, pour une équation différentielle à une dimension. La courbe noire correspond à la solution du problème de Cauchy, les points rouges sont les approximations numériques fournies par un schéma classique (Euler par exemple), et les encadrements marrons représentent les formes géométriques recherchées par l'intégration garantie. En particulier, la recherche d'encadrements globaux (rectangles bleus en tirets sur la figure 3.1) est intéressante pour l'analyse statique des systèmes hybrides.

Formellement, le but d'un algorithme d'intégration garantie est donc de trouver une suite de couples $\left(t_n, [\boldsymbol{y_n}]\right)_{n\in\mathbb{N}}$, où $t_n \in \mathbb{R}_+$ et $[\boldsymbol{y_n}] \in I_{\mathbb{F}}^n$ telle que

$$\forall n \in \mathbb{N},\ y_\infty\left(t_n\right) \in [\boldsymbol{y_n}]\ .$$

En pratique, on se limitera tout le temps à trouver des suites finies $\left(t_n, [\boldsymbol{y_n}]\right)_{0\leq n\leq N}$. Un deuxième intérêt de l'intégration garantie est de pouvoir gérer des problèmes de Cauchy où la condition initiale n'est pas précisément connue mais seulement donnée par un intervalle : $y(0) \in [y_0]$. Dans ce cas, on veut pouvoir donner des encadrements de l'ensemble des solutions à un problème de Cauchy donné par une condition initiale $y(0) = y_0$, avec $y_0 \in [y_0]$.

Problème 3.1 (Intégration garantie) *Étant donné une équation différentielle de dimension n $\dot{y} = F(y)$, une condition initiale donnée par $y(0) \in [y_0]$ et un temps final T, trouver une suite de couples $\left(t_n, [\boldsymbol{y_n}]\right)_{0\leq n\leq N}$ telle que :*

- *$t_0 = 0$, $t_i \leq t_{i+1}$ et $t_N = T$;*
- *toute solution y_∞ à un problème de Cauchy $\dot{y} = F(y),\ y(0) = y_0$ avec $y_0 \in [y_0]$ vérifie*

$$\forall n \in [\![0, N]\!],\ y_\infty(t_n) \in [\boldsymbol{y_n}]\ .$$

Le problème de l'intégration garantie globale s'exprime de la même façon, sauf que les encadrements doivent vérifier $\forall t \in [t_n, t_{n+1}],\ y_\infty(t) \in [\boldsymbol{y_n}]$.

Remarque (1) Le cas des équations différentielles à paramètres ($\dot{y} = F(y,k)$) où k n'est pas connu exactement ($k \in [k]$) est inclus dans notre formalisme. Il suffit de considérer k comme une variable continue constante (dont la dérivée est nulle) ayant comme valeur initiale l'intervalle $[k]$. Cependant, des méthodes plus avancées pour ce type de problèmes existent. Elles reposent essentiellement sur l'utilisation des modèles de Taylor [LS07]. Ces techniques dépassent le cadre de cette thèse.

Remarque (2) À partir de maintenant, nous ne considérerons que le cas d'équations différentielles de dimension 1, les dimensions supérieures sont une extension triviale du cas de dimension 1.

3.3 Méthode des séries de Taylor

La première solution apportée au problème 3.1 est une méthode reposant sur les séries de Taylor [NJC99, Lö88, Eij81] ; elle remonte aux débuts de l'arithmétique d'intervalle de Moore [Moo66]. Il s'agit en fait de la première application de l'arithmétique d'intervalle. Cette méthode est encore la plus utilisée aujourd'hui, elle est notamment au cœur de outils AWA [Lö88] et VNODE [NJ01a]. C'est une méthode itérative explicite (elle calcule $[\boldsymbol{y_{n+1}}]$ en fonction de $[\boldsymbol{y_n}]$ uniquement) qui possède deux étapes :

- une étape de prédiction/validation qui calcule t_{n+1} et un encadrement grossier de y_∞ sur $[t_n, t_{n+1}]$, validant ainsi l'existence de la solution sur $[t_n, t_{n+1}]$;
- une étape de correction/raffinement qui diminue l'encadrement a priori pour donner une garantie précise sur y_∞ au point t_{n+1}.

3.3.1 Un méthode en deux étapes

Le calcul de l'encadrement $[\boldsymbol{y_{n+1}}]$ est obtenu à partir de $[\boldsymbol{y_n}]$ de la manière suivante. On commence par décomposer une solution au problème de Cauchy en série de Taylor jusqu'à l'ordre N (on utilise le théorème de Taylor avec reste de Lagrange) :

$$\exists t' \in [t_n, t_{n+1}] \ : \ y_\infty(t_{n+1}) = y_\infty(t_n) + \sum_{i=1}^{N-1} \frac{h_{n+1}^i}{i!} \frac{d^i y_\infty}{dt^i}(t_n) + \frac{h_{n+1}^N}{N!} \frac{d^N y_\infty}{dt^N}(t') \ . \tag{3.5}$$

Dans cette équation, $h_{n+1} = t_{n+1} - t_n$ représente le pas d'intégration choisi. Nous allons utiliser l'équation (3.5) pour obtenir un encadrement de $y_\infty(t_{n+1})$. Commençons par les termes simples. Par hypothèse, on sait que $y_\infty(t_n) \in [\boldsymbol{y_n}]$, donc on a :

$$\exists t' \in [t_n, t_{n+1}] \ : \ y(t_{n+1}) \in [\boldsymbol{y_n}] + \sum_{i=1}^{N-1} \frac{h_{n+1}^i}{i!} \frac{d^i y_\infty}{dt^i}(t_n) + \frac{h_{n+1}^N}{N!} \frac{d^N y_\infty}{dt^N}(t') \ . \tag{3.6}$$

Par ailleurs, on sait que $\dot{y} = F(y)$. Soit $F^{[n]}$ la suite de fonctions définies par $F^{[1]} = F$ et $F^{[i]} = \frac{F}{i} \frac{\partial F^{[i-1]}}{\partial y}$ pour $i \geq 2$. Alors, il est aisé de démontrer (par récurrence) que

$$\forall i \in [\![1, N]\!], \ \frac{1}{i!} \frac{d^i y_\infty}{dt^i}(t_n) = F^{[i]}\big(y_\infty(t_n)\big) \ .$$

En injectant ces égalités dans l'équation (3.6) et en utilisant l'encadrement $y_\infty(t_n) \in [\boldsymbol{y_n}]$, on obtient :

$$\exists t' \in [t_n, t_{n+1}] \ : \ y_\infty(t_{n+1}) \in [\boldsymbol{y_n}] + \sum_{i=1}^{N-1} h_{n+1}^i F^{[i]}([\boldsymbol{y_n}]) + h_{n+1}^N F^{[N]}\big(y_\infty(t')\big) \ . \tag{3.7}$$

Dans l'équation (3.7), il reste un terme à borner pour obtenir un encadrement de $y(t_{n+1})$. Ce terme nécessite de calculer un intervalle $[\widetilde{\boldsymbol{y_n}}]$ tel que $\forall t \in [t_n, t_{n+1}]\,, \ y_\infty(t) \in [\widetilde{\boldsymbol{y_n}}]$. La première

étape de l'algorithme d'intégration garantie sera donc le calcul de cet encadrement. Si on insère ensuite $[\widetilde{\boldsymbol{y_n}}]$ dans l'équation (3.7), on obtient la formule suivante pour $[\boldsymbol{y_{n+1}}]$:

$$[\boldsymbol{y_{n+1}}] = [\boldsymbol{y_n}] + \sum_{i=1}^{N-1} h_{n+1}^i F^{[i]}([\boldsymbol{y_n}]) + h_{n+1}^N F^{[N]}([\widetilde{\boldsymbol{y_n}}]) \ . \tag{3.8}$$

Cependant, en utilisant cette formule on voit que la largeur de l'intervalle $[\boldsymbol{y_{n+1}}]$ est supérieur à la largeur de $[\boldsymbol{y_n}]$, et donc cette formule donne une méthode très instable. La seconde étape de l'algorithme d'intégration garantie sera donc de calculer un encadrement le plus fin possible pour $[\boldsymbol{y_{n+1}}]$.

3.3.2 Encadrement a priori

La première étape doit donc calculer, étant donnés $[\boldsymbol{y_n}]$, t_n et t_{n+1}, un intervalle $[\widetilde{\boldsymbol{y_n}}]$ tel que, si $y(t_n) \in [\boldsymbol{y_n}]$, alors $\forall t \in [t_n, t_{n+1}]\,,\ y(t) \in [\widetilde{\boldsymbol{y_n}}]$. Si plusieurs méthodes existent pour cela (encadrement constant [Eij81, Lö88], polynomial [Lö95] ou d'ordre supérieur [Moo66]), toutes reposent sur l'application de l'opérateur de Picard-Lindelöf sur le bon espace de fonctions puis l'utilisation du théorème de point fixe de Banach.

Définition 3.6 (Opérateur de Picard-Lindelöf) Soit $\mathcal{D} \subseteq \mathcal{C}^0$ un ensemble de fonctions continues définies sur un voisinage de t_n. L'opérateur de Picard-Lindelöf transforme une fonction continue en une autre fonction continue donnée par :

$$\Psi : \left\{ \begin{array}{l} \mathcal{D} \to \mathcal{C}^0 \\ f \mapsto \lambda t.y_\infty(t_n) + \int_{t_n}^t F(f(s))ds \end{array} \right. . \tag{3.9}$$

D'après le théorème de Picard 3.2, le problème de Cauchy possède une unique solution si et seulement si Ψ possède un point fixe. Nous allons utiliser le théorème du point fixe de Banach pour prouver son existence.

Théorème 3.5 (Théorème de Banach.) *Soit $\Phi : Y \to Y$ une fonction contractante définie sur un espace métrique Y. Alors, Φ possède un unique point fixe $y^* \in Y$, c'est-à-dire que $\Phi(y^*) = y^*$.*

Nous décrivons maintenant la méthode la plus simple, dite de l'encadrement constant. Cette méthode (comme toutes les autres) est une méthode *validante* : elle peut réussir, auquel cas le pas d'intégration de t_n à t_{n+1} est validé, ou échouer, auquel cas le pas d'intégration est rejeté et un nouveau pas doit être choisi. Un enjeu important des méthodes d'intégration garantie est de limiter le nombre de pas rejetés, et les méthodes d'ordre supérieur permettent justement cela [Lö95, Moo66, NJP01]. La méthode à encadrement constant repose sur le théorème suivant.

Théorème 3.6 *Soit $[\boldsymbol{R}] \in I_{\mathbb{F}}$ tel que $[\boldsymbol{y_n}] \subseteq [\boldsymbol{R}]$. Si $[\boldsymbol{y_n}] + [0, h_n]\, F\big([\boldsymbol{R}]\big) \subseteq [\boldsymbol{R}]$, alors il existe une unique solution y au problème de Cauchy sur l'intervalle $[t_n, t_{n+1}]$ et cette solution vérifie*

$$\forall t \in [t_n, t_{n+1}, ,]\ \ y_\infty(t) \in [\boldsymbol{R}] \ .$$

Preuve Soit $[\boldsymbol{R}]$ un intervalle contenant $[\boldsymbol{y_n}]$, et soit Y l'ensemble des fonctions continues définies sur $[t_n, t_{n+1}]$ à valeur dans $[\boldsymbol{R}]$. Soit $f \in Y$, on a alors :

$$\begin{array}{rcl} (\Psi f)(t) & = & y_\infty(t_n) + \displaystyle\int_{t_n}^t F\big(f(s)\big)ds \\ & \in & y_\infty(t_n) + \displaystyle\int_{t_n}^t F\big([\boldsymbol{R}]\big)ds \\ & \in & [\boldsymbol{y_n}] + [0, h_n]\,[\boldsymbol{R}] \end{array}$$

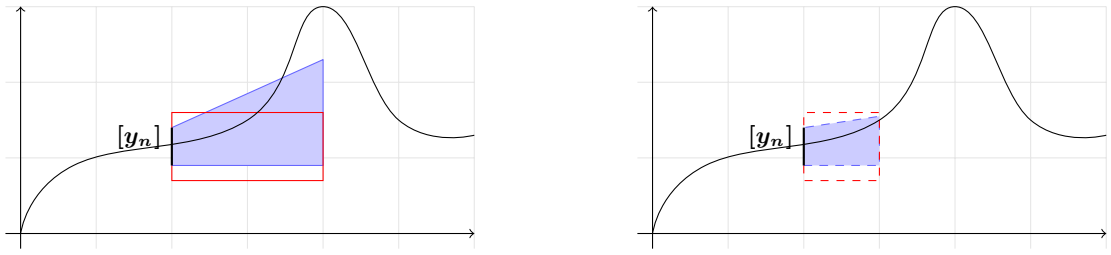
D'après les hypothèses du théorème, on a donc $\forall t \in [t_n, t_{n+1}]\,,\ (\Psi f)(t) \in [\boldsymbol{R}]$, donc $\Psi f \in Y$. Ψ est donc une fonction contractante (pour la norme exponentielle) de Y dans lui même. D'après le théorème de Banach, elle possède donc un point fixe y_∞, qui, d'après le théorème de Picard 3.2, est l'unique solution au problème de Cauchy sur $[t_n, t_{n+1}]$. □

Le théorème 3.6 est à la base de l'algorithme de calcul de l'encadrement a priori par la méthode constante (algorithme 1). Intuitivement, la méthode fonctionne ainsi : on choisit arbitrairement un premier encadrement que l'on utilise pour faire une extrapolation garantie à partir de la méthode d'Euler (calcul de $[\boldsymbol{y_n}] + [0, h_n]\, F([\boldsymbol{R}])$). Si cette extrapolation dépasse $[\boldsymbol{R}]$ (figure 3.2(a)), on refuse le pas et la méthode sera ré-appliquée avec un pas plus petit. Si au contraire l'extrapolation reste dans $[\boldsymbol{R}]$ (figure 3.2(b)), on valide le pas et on choisit $[\boldsymbol{R}]$ pour $[\widetilde{\boldsymbol{y_n}}]$.

Entrée : $[\boldsymbol{y_n}]$; `/* encadrement à` t_n `*/`
Entrée : h_n ; `/* pas d'intégration choisi */`
Résultat : $[\widetilde{\boldsymbol{y_n}}]$ tel que $\forall t \in [t_n, t_n + h_n]\,,\ y(t) \in [\widetilde{\boldsymbol{y_n}}]$.

début
  Trouve $[\widetilde{\boldsymbol{y_n}}] \supset [\boldsymbol{y_n}]$;
  Calcule $[\widetilde{\boldsymbol{y_n}}]' = [\boldsymbol{y_n}] + [0, h_n]\, F([\widetilde{\boldsymbol{y_n}}])$;
  si $[\widetilde{\boldsymbol{y_n}}]' \subseteq [\widetilde{\boldsymbol{y_n}}]$ **alors**
    renvoyer $[\widetilde{\boldsymbol{y_n}}]$
  sinon
    echoue
  finsi
fin

Algorithme 1 : Validation d'un pas de calcul par Picard-Lindelöf.



(a) Pas non validé. (b) Pas validé.

FIG. 3.2 – Utilisation de l'opérateur de Picard-Lindelöf.

3.3.3 Réduction de l'encadrement

Après la validation du pas d'intégration et le calcul de l'encadrement $[\boldsymbol{y_n}]$, on peut utiliser l'équation (3.8) pour trouver un encadrement de $y_\infty(t_{n+1})$. Cependant, cette formule est intrinsèquement mauvaise car elle ne peut que faire grossir la largeur des intervalles. En effet, $[\boldsymbol{y_{n+1}}]$ est défini comme la somme de $[\boldsymbol{y_n}]$ et d'un autre intervalle. D'après les règles du calcul par intervalle, la largeur du résultat d'une somme est supérieure à la largeur des deux opérandes. La largeur de $[\boldsymbol{y_{n+1}}]$ est donc supérieure à celle de $[\boldsymbol{y_n}]$. Pour empêcher cela, nous allons effectuer les calculs d'intervalles en utilisant la forme centrée [JKDW01]. Nous représentons donc $[\boldsymbol{y_n}]$ comme un point $\widehat{\boldsymbol{y_n}}$, le milieu de $[\boldsymbol{y_n}]$, et un petit intervalle $[\boldsymbol{r_n}]$ centré en 0 tel que $[\boldsymbol{y_n}] = \widehat{\boldsymbol{y_n}} + [\boldsymbol{r_n}]$. On utilise alors le théorème 3.7 pour calculer de manière plus précise les fonctions d'inclusions.

Théorème 3.7 (Calcul d'une fonction d'inclusion par forme centrée.) *Soit* $[\boldsymbol{R}] = \widehat{\boldsymbol{R}} + [\boldsymbol{r}]$ *un intervalle donnée sous forme centrée et* f *une fonction continûment dérivable sur* R*. On définit la fonction* f_c *par :*

$$f_c\big([\boldsymbol{R}]\big) = f(\widehat{\boldsymbol{R}}) + J\big(f, [\boldsymbol{R}]\big).[\boldsymbol{r}] \ . \tag{3.10}$$

On a alors :

$$\{f(x) \ : \ x \in [\boldsymbol{R}]\} \subseteq f_c\big([\boldsymbol{R}]\big) \ .$$

Le théorème 3.7 permet donc de calculer une fonction d'inclusion qui est souvent (mais pas toujours) plus précise que la fonction d'inclusion obtenue en remplaçant chaque opérateur par son équivalent dans l'arithmétique d'intervalle (voir [JKDW01] pour une comparaison poussée entre ces deux modes de calcul). Si on injecte le théorème 3.7 dans l'équation (3.8), on obtient :

$$\begin{array}{rcl} [\boldsymbol{y_{n+1}}] & = & \widehat{\boldsymbol{y_n}} + [\boldsymbol{r_n}] + \\ & & \sum_{i=1}^{N-1} h_{n+1}^i F^{[i]}(\widehat{y_n}) + \Big(\sum_{i=1}^{N-1} h_{n+1}^i J\big(F^{[i]}, [\boldsymbol{y_n}]\big)\Big)\,([\boldsymbol{r_n}]) + \\ & & h_{n+1}^N F^{[N]}([\widehat{\boldsymbol{y_n}}]) \end{array} \ . \tag{3.11}$$

En utilisant l'équation (3.11), on peut montrer que, si le problème de Cauchy sous-jacent est contractant, alors la méthode est stable : la largeur de l'intervalle $[\boldsymbol{y_{n+1}}]$ serait inférieure à la largeur de l'intervalle $[\boldsymbol{y_n}]$ si on pouvait calculer de manière exacte chacun des termes intervenant dans l'équation (3.11). En pratique, le calcul peut se révéler instable à cause de l'*effet enveloppant*, ou *wrapping effect* en anglais [JKDW01]. Le *wrapping effect* intervient dans tout calcul sur des rectangles de dimension $n \geq 2$. Soit $R \in I_{\mathbb{D}}^n$ un rectangle de dimension n, et soit f une fonction continue que l'on souhaite évaluer sur R. On veut donc calculer $R' = \{f(x) \ : \ x \in R\}$. Pour calculer R', on utilise l'arithmétique d'intervalle et donc une fonction d'inclusion à la place de f. On obtiendra donc une surapproximation $\widetilde{R}$ de R' qui peut être très large (voir la figure 3.3) : si R' a une forme très éloignée d'un rectangle, la surapproximation est forcément très forte ; dans le cas de la figure 3.3, le rectangle rouge représente la meilleure approximation possible $\widetilde{R}$ de R'.

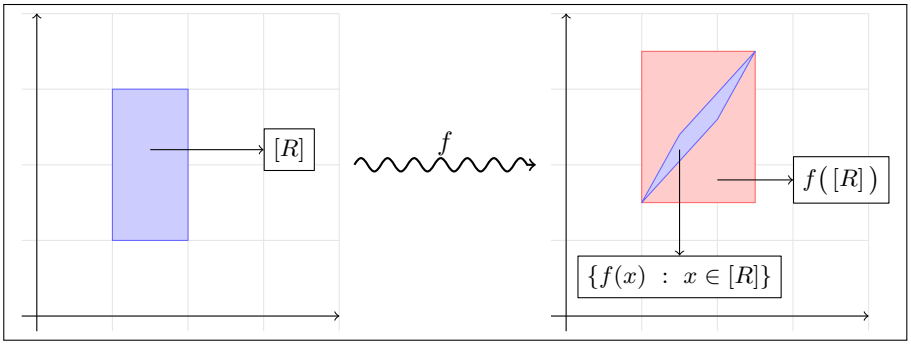


FIG. 3.3 – Exemple de wrapping effect.

3.3.4 Limitation du wrapping effect

Le wrapping effect est certainement le principal facteur d'imprécision des méthodes d'intégration garantie [NJ01b]. Dans la formule (3.11), il intervient lors de la multiplication matricielle entre $J\big(F^{[i-1]}, [\boldsymbol{y_n}]\big)$ et $[\boldsymbol{r_n}]$ si, par exemple, la matrice jacobienne est une matrice de rotation, alors le wrapping effect sera important. Pour réduire ce phénomène, plusieurs méthodes ont été proposées par Moore [Moo66], Eijgenraam [Eij81], avec plus ou moins de succès [NJC99]. Nous décrivons ici rapidement la méthode de Löhner dite de "factorisation QR" et l'illustrons par des schémas explicatifs (empruntés à [NJC99]).

Dans la formule (3.11), notons

$$[\boldsymbol{z_{n+1}}] = h_{n+1}^N F^{[N]}([\widetilde{\boldsymbol{y_n}}]) \tag{3.12}$$
$$\boldsymbol{s_{n+1}} = mid\big([\boldsymbol{z_{n+1}}]\big) \tag{3.13}$$
$$\widehat{\boldsymbol{y_{n+1}}} = \widehat{y_n} + \sum_{i=1}^{N-1} h_{n+1}^i F^{[i]}(\widehat{y_n}) + \boldsymbol{s_{n+1}} \tag{3.14}$$
$$[\boldsymbol{S_n}] = I + \sum_{i=1}^{N-1} h_{n+1}^i J\big(F^{[i]}, [\boldsymbol{y_n}]\big) \tag{3.15}$$

En utilisant ces notations et l'équation (3.11), on obtient (avec $A_0 = I_N$, la matrice identité de dimension N) :

$$y(t_1) \in \widehat{\boldsymbol{y_1}} + \big([\boldsymbol{S_0}]A_0\big)[\boldsymbol{r_0}] + [\boldsymbol{z_1}] - s_1 = \big[\boldsymbol{y_1^1}\big] \tag{3.16}$$
$$y(t_1) \in \widehat{\boldsymbol{y_1}} + A_1.\big(A_1^{-1}([\boldsymbol{S_0}]A_0)[\boldsymbol{r_0}] + A_1^{-1}([\boldsymbol{z_1}] - z_1\big) = \widehat{\boldsymbol{y_1}} + A_1[\boldsymbol{q_1}] = \big[\boldsymbol{y_1^2}\big] \tag{3.17}$$
$$[\boldsymbol{y_1}] = \big[\boldsymbol{y_1^1}\big] \cap \big[\boldsymbol{y_1^2}\big] \tag{3.18}$$

On définit donc l'encadrement à t_1 comme l'intersection de deux intervalles qui sont obtenus d'une part en évaluant (3.11) directement (équation (3.16)), et d'autre part en effectuant avant le calcul de $[\boldsymbol{S_0}][\boldsymbol{r_0}]$ un changement de base via une matrice régulière A_1 (équation (3.20)). Le choix de cette matrice est primordial pour que le wrapping effect soit limité, et nous le détaillons dans la suite. Expliquons d'abord comment, à partir de A_1, $[\boldsymbol{y_1}]$ et $[\boldsymbol{q_1}] = \big(A_1^{-1}([\boldsymbol{S_0}]A_0)[\boldsymbol{r_0}] + A_1^{-1}([\boldsymbol{z_1}] - z_1\big)$ on obtient un encadrement $[\boldsymbol{y_2}]$ de $y(t_2)$. Il suffit de répéter la même opération que pour $[\boldsymbol{y_1}]$:

$$y(t_2) \in \widehat{\boldsymbol{y_2}} + \big([\boldsymbol{S_1}]A_1\big)[\boldsymbol{q_1}] + [\boldsymbol{z_2}] - s_2 = \big[\boldsymbol{y_2^1}\big] \tag{3.19}$$
$$y(t_2) \in \widehat{\boldsymbol{y_2}} + A_2.\big(A_2^{-1}([\boldsymbol{S_1}]A_1)[\boldsymbol{q_1}] + A_2^{-1}([\boldsymbol{z_2}] - z_2\big) = \widehat{\boldsymbol{y_2}} + A_2[\boldsymbol{q_2}] = \big[\boldsymbol{y_2^2}\big] \tag{3.20}$$
$$[\boldsymbol{y_2}] = \big[\boldsymbol{y_2^1}\big] \cap \big[\boldsymbol{y_2^2}\big] \tag{3.21}$$

Encore une fois, il est important de bien choisir la matrice régulière A_2. En répétant cette méthode, on obtient l'algorithme 2 qui calcule $[\boldsymbol{y_{n+1}}]$, S_{n+1} et $[\boldsymbol{q_{n+1}}]$ à partir de $[\boldsymbol{y_n}]$, A_n et $[\boldsymbol{q_n}]$.

Entrée : $[\boldsymbol{y_n}]$; /* encadrement à t_n */
Entrée : h_n ; /* pas d'intégration choisi */
Entrée : $\widehat{\boldsymbol{y_n}}$, $[\widetilde{\boldsymbol{y_n}}]$, A_n, $[\boldsymbol{q_n}]$
Résultat : $\widehat{\boldsymbol{y_{n+1}}}$, $[\boldsymbol{y_{n+1}}]$, A_{n+1}, $[\boldsymbol{q_{n+1}}]$

début
- $[\boldsymbol{z_{n+1}}] = h_{n+1}^N F^{[N]}([\widetilde{\boldsymbol{y_n}}])$;
- $\boldsymbol{s_{n+1}} = mid\big([\boldsymbol{z_{n+1}}]\big)$;
- $\widehat{\boldsymbol{y_{n+1}}} = \widehat{y_n} + \sum_{i=1}^{N-1} h_{n+1}^i F^{[i]}(\widehat{y_n}) + \boldsymbol{s_{n+1}}$;
- $[\boldsymbol{S_n}] = I + \sum_{i=1}^{N-1} h_{n+1}^i J\big(F^{[i]}, [\boldsymbol{y_n}]\big)$;
- Choisi A_{n+1};
- $[\boldsymbol{q_{n+1}}] = \big(A_{n+1}^{-1}([\boldsymbol{S_n}]A_n)[\boldsymbol{q_n}] + A_{n+1}^{-1}([\boldsymbol{z_{n+1}}] - s_{n+1}\big)$;
- $\Big[\boldsymbol{y_{n+1}^1}\Big] = \widehat{\boldsymbol{y_{n+1}}} + \big([\boldsymbol{S_n}]A_n\big)[\boldsymbol{q_n}] + [\boldsymbol{z_{n+1}}] - s_{n+1}$;
- $\Big[\boldsymbol{y_{n+1}^2}\Big] = \widehat{\boldsymbol{y_{n+1}}} + A_{n+1}[\boldsymbol{q_{n+1}}]$;
- $[\boldsymbol{y_{n+1}}] = \Big[\boldsymbol{y_{n+1}^1}\Big] \cap \Big[\boldsymbol{y_{n+1}^2}\Big]$;

fin

Algorithme 2 : Réduction de l'encadrement par la méthode de Löhner.

Comme nous l'avons déjà dit, c'est le choix de A_{n+1} qui rend la méthode intéressante pour la diminution du wrapping effect. Dans la méthode de factorisation QR de Löhner, le choix s'effectue

comme suit. On calcule d'abord $\widehat{A_{n+1}} = mid([\boldsymbol{S_n}]A_n)$ puis on effectue la décomposition QR de $\widehat{A_{n+1}}$: $\widehat{A_{n+1}} = Q_{n+1}R_{n+1}$ où Q_{n+1} est une matrice orthogonale et R_{n+1} est une matrice triangulaire supérieure. On pose alors $A_{n+1} = Q_{n+1}$. Intuitivement, cela revient à changer le repère orthogonal dans lequel on effectue le calcul $([\boldsymbol{S_n}]A_n)[\boldsymbol{q_n}]$ de telle sorte qu'un des axes du nouveau repère soit parallèle à une des arête du parallélépipède définie par $\left\{\widehat{A_{n+1}}.q \ : \ q \in [\boldsymbol{q_n}]\right\}$. Cela se traduit par un meilleur encadrement du résultat que dans le repère initial (figure 3.4, ligne 1 et 2). Un choix encore meilleur consiste à permuter dans un premier temps les colonnes de $\widehat{A_{n+1}}$ de telle sorte que la première colonne corresponde à la plus grande arrête du parallélépipède $\left\{\widehat{A_{n+1}}.q \ : \ q \in [\boldsymbol{q_n}]\right\}$. Ainsi, dans le repère engendré par Q_{n+1}, un des axes est parallèle à cette arête, ce qui améliore encore le résultat (figure 3.4, troisième ligne). On pose alors $A_{n+1} = Q_{n+1}$, et on obtient la méthode QR de Löhner.

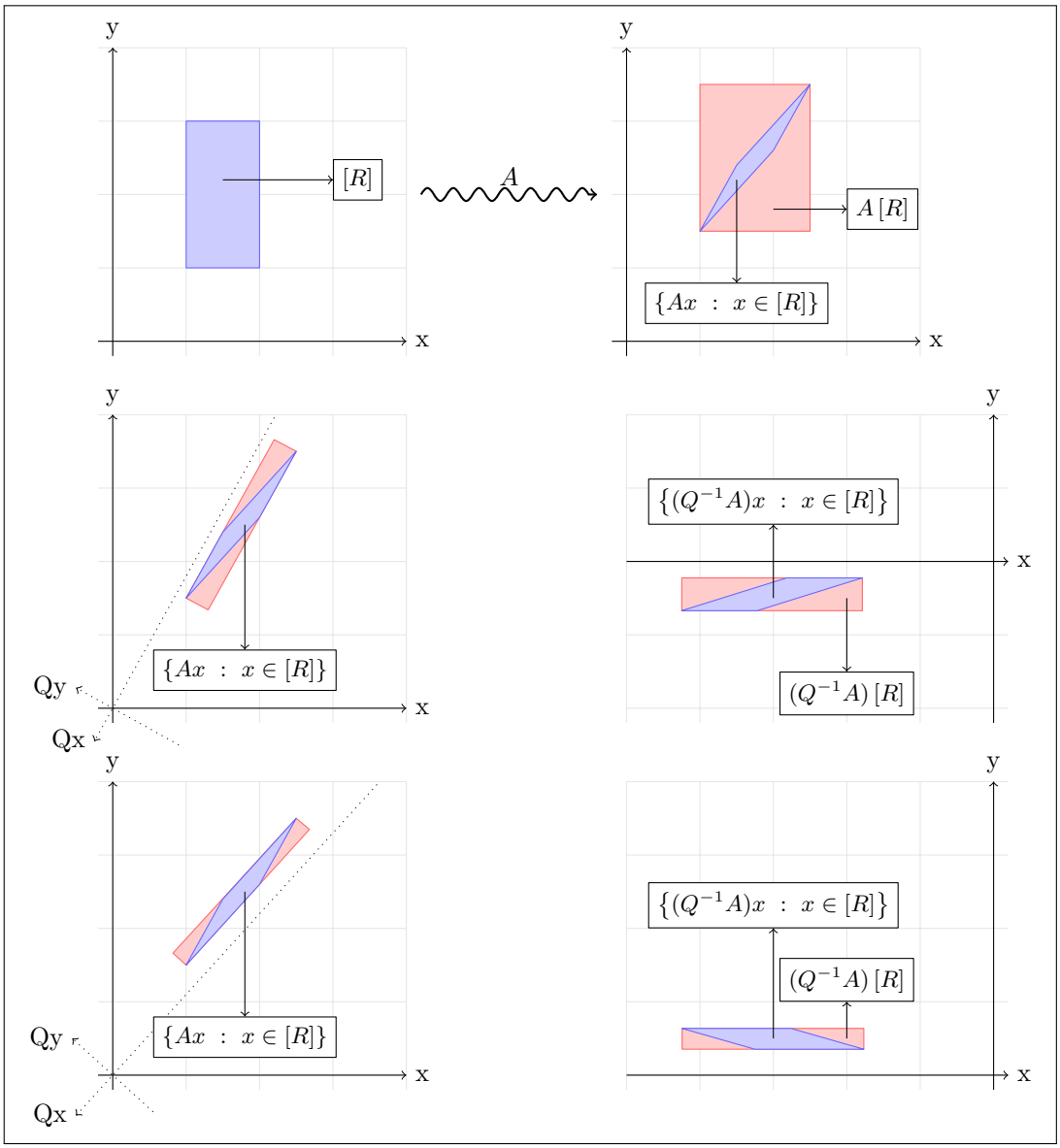


FIG. 3.4 – Réduction du wrapping effect par décomposition QR.

	Encadrement de h_1	Temps CPU (s)
CI1	6.25144769981[78620,80300]	0.12
CI2	6.251[3591546711086,5362449647825]	0.14
Paramètres intervalle	[6.1890431083846806,6.2515002772193374]	0.3

TAB. 3.1 – Efficacité de VNODE sur le problème (3.22).

3.4 Outils

Nous présentons ici rapidement les outils les plus connus pour la résolution garantie d'équations différentielles, en insistant sur leurs différences. Pour un comparatif plus détaillé, avec notamment des comparaisons d'efficacité, on pourra se référer à [Ned06]. Nous montrons pour chaque outil un programme qui permet la résolution garantie de l'équation différentielle du système des deux réservoirs introduit à la section 1.6, dans le cas où les deux vannes sont ouvertes et le niveau d'eau du deuxième réservoir inférieur à la hauteur H :

$$\begin{cases} \dot{h_1} &= i - k_1\sqrt{h1} \\ \dot{h_2} &= k_1\sqrt{h_1} - k_2\sqrt{h_2} \end{cases} . \tag{3.22}$$

Nous étudierons ce problème avec, dans un premier temps, des conditions initiales (CI1) données par un singleton : $h_1 = 10, h_2 = 4$ et les paramètres $k_1 = 2$, $k_2 = 3$ et i =5. Ensuite, nous calculerons un encadrement garanti pour des conditions initiales avec incertitude (CI2) : $h_1 \in [9.8, 10.2]$ et $h_2 \in [4, 4.01]$. Enfin, nous utiliserons les conditions initiales ponctuelles et nous imposerons une incertitude sur les paramètres de l'EDO : $k_1 \in [2, 2.01]$ et $k_2 \in [3, 3.1]$. À chaque fois le temps d'intégration sera de $T = 20$.

3.4.1 VNODE

VNODE[1] [NJ01a, NJ99] est une bibliothèque C++ d'intégration garantie développée par Ned Nedialkov. Elle repose sur une méthode de Hermite-Obreschkoff par intervalle et un développement en séries de Taylor de la solution de l'équation différentielle. Elle utilise les techniques avancées pour calculer l'encadrement a priori et pour limiter le wrapping effect. Grâce à l'utilisation des *templates* C++ [VJ02] elle est très simple d'utilisation, notamment dans sa dernière version VNODELP. Le calcul des dérivées successives de la solution est fait par différentiation automatique [Ral81, Gri00, BHN02] grâce à la librairie FADBAD [BS96] et les calculs d'intervalle sont gérés par la librairie PROFIL/BIAS [Knu94]. Un programme permettant de résoudre l'équation (3.22) est donné à la figure 3.5. Le tableau 3.1 donne les résultat obtenus par VNODE pour l'équation (3.22).

3.4.2 Cosy

COSY Infinity [2] [BM06a, BM98] est un système de calcul scientifique initialement développé pour la physique des particules. Des composants de calcul garanti utilisant les modèles de Taylor [BH98] y ont été ajoutés, et notamment un intégrateur garanti nommé COSY-VI (*COSY Verified Integrator*). L'algorithme d'intégration de COSY-VI reprend les méthodes que nous avons mentionnées, mais utilise une représentation des données sous forme de modèles de Taylor à la place des intervalles. Les modèles de Taylor permettent de représenter des ensembles de points de manière plus précise que les intervalles (et notamment des ensembles de points non convexes) et souffrent ainsi moins du wrapping effect. Cette utilisation des modèles de Taylor peut être vue comme un développement en séries de Taylor de la solution en fonction du temps et des conditions initiales. Le système COSY-VI est écrit en Fortran, avec une interface C++, mais reste d'usage assez complexe. Tous les calculs d'intervalles sont fait par COSY Infinity, et leur correction vis-à-vis des problèmes liés à l'arithmétique des ordinateurs a été prouvée [RMB05]. Le principal défaut

[1] `http ://www.cas.mcmaster.ca/~nedialk/Software/VNODE/VNODE.shtml`
[2] `http ://bt.pa.msu.edu/index_cosy.htm`

```
template<typename var_type>  void equation(int n,var_type*yp,
   const var_type*y,
   var_type t,void*param){
  interval k1(3,3.001),k2(2,2.001),i(5,5.001);
  yp[0] = i-k1*sqrt(y[0]);
  yp[1] = k1*sqrt(y[0])-k2*sqrt(y[1]);
}

int main(){
  /* condition initiale */
  iVector h(2);  h[0]= 10;   h[1]= 4;
  /* temps */
  interval t0 (0),tend(20);
  AD*ad= new FADBAD_AD(2,equation,equation);
  VNODE*Solver= new VNODE(ad);
  Solver->integrate(t0,h,tend);
  cout<<"Solution enclosure at t = "<< t0 <<endl;
  printVector(h);
  return 0;
}
```

FIG. 3.5 – Programme VNODE pour résoudre le problème (3.22).

	Encadrement de h_1	Temps CPU
CI1	6.2514476998179[3,6]	0.44
CI2	-	-
Paramètres intervalle	-	-

TAB. 3.2 – Efficacité de AWA sur le problème (3.22).

de COSY, outre son utilisation laborieuse, est sa lenteur : du fait de l'utilisation des modèles de Taylor, le processus d'intégration est nettement plus lent. En revanche, la stabilité de la solution calculée est très bonne, même en présence d'incertitudes élevées sur les conditions initiales.

Remarque Pour des problèmes de brevets et de conflits entre l'université du Michigan et SUN Microsystems, l'outil d'intégration garantie de COSY n'est plus disponible librement. Nous n'avons donc pas pu le tester et les remarques précédentes sur son utilisation viennent de discussion avec des utlisateurs réguliers de COSY. Pour ces raisons nous ne montrons donc pas les temps de calcul de COSY sur le problème (3.22).

3.4.3 AWA

AWA[3] [Lö88] est un des plus anciens intégrateurs garantis. Son implémentation est très proche des techniques que nous avons détaillées ici. Écrit en Pascal, il est d'installation et d'utilisation facile. Cependant, cela le rend également très difficilement intégrable dans un projet plus vaste et le temps de calcul est élevé comparé aux outils plus récents. Il n'est pas possible avec AWA de donner une condition initiale intervalle ni des paramètres intervalles dans les équations différentielles, nous ne donnons donc les résultats de AWA que pour la condition initiale CI1. Le programme permettant de calculer ces résultats avec AWA est donné à la figure 3.6.

[3] `http ://www.math.uni-wuppertal.de/~xsc/xsc/pxsc_software.html`

```
F1 = 5-3*sqrt(U1)
F2 = 3*sqrt(U1) - 2*sqrt(U2)
;;

1 0 20 0 18

4 0

10 4

1e-16 1e-16

n n n
```

FIG. 3.6 – Programme AWA pour résoudre le problème (3.22).

CHAPITRE 4

Systèmes hybrides

Les systèmes hybrides sont des systèmes qui présentent des comportements discrets (changement instantané et non continu d'état) et des comportements continus (évolution d'un état vers un autre en suivant un flot continu dans le temps). Ils sont donc à l'intersection de deux communautés qui ont chacune développé leur propre formalisme pour les étudier. Du point de vue de la théorie des systèmes dynamiques et du contrôle, un système hybride est avant tout un système continu étendu par des fonctions de reset. La modélisation des systèmes hybrides se fait alors via des fonctions affines par morceaux (*piecewise affine systems*, PWA), via des systèmes mixant logique et dynamique (*mixed logic dynamical systems*, MLD) ou encore via des équations différentielles discontinues. Heemels et al. formalisent ces modèles et proposent une comparaison de leurs pouvoirs expressifs ainsi que des techniques de vérification [HDB01]. D'un point de vue plus informatique, un système hybride est essentiellement un système à évènements discrets étendu par des équations différentielles permettant de modéliser les évolutions des variables continues. Ainsi, la théorie des automates finis et des algèbres de processus ont leur équivalent hybride. Nous nous plaçons dans cette optique et présentons ici les modèles de systèmes hybrides les plus "informatique".

4.1 Exemple introductif

Avant de formaliser la théorie des systèmes hybrides, et celle des automates hybrides en particulier, nous allons introduire les concepts hybrides par un exemple classique. Notre formalisation est légèrement différente de [Hen96] mais reste équivalente.

Description du système. Le système hybride très simple que nous allons étudier est celui du thermostat [Hen96], dont nous avons juste changé les valeurs des paramètres : un radiateur est placée dans une pièce, et est relié à un thermostat qui est chargé de maintenir la température de la pièce entre 19 et 26 degrés. Pour cela, le thermostat mesure à tout instant la température, et si elle passe au-dessus (respectivement au-dessous) de 19 (respectivement 26) degrés, il allume (respectivement éteint) le radiateur. Clairement, quand le radiateur est allumé, la température de la pièce augmente, et quand il est éteint, la température diminue. Nous avons bien affaire à un système hybride composé :

- d'une partie continue (température x de la pièce) qui comporte deux modes : $\dot{x} = 9 - x/3$ si le radiateur est allumé, $\dot{x} = -x/3$ si le radiateur est éteint ;
- et d'une partie discrète, le thermostat qui possède un capteur mesurant x et un actionneur pour allumer/éteindre le radiateur.

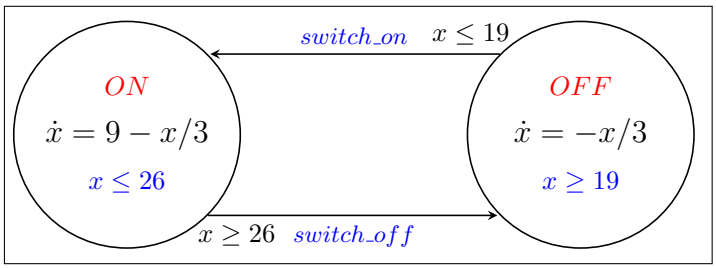


FIG. 4.1 – Automate hybride représentant le système du thermostat.

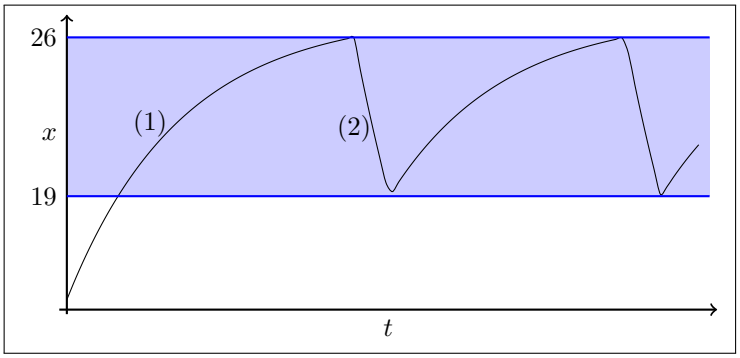


FIG. 4.2 – Évolution de la température x lors d'une exécution de l'automate du thermostat simple.

L'automate équivalent. Une façon de représenter ce système hybride est l'automate hybride de la figure 4.1. Ce modèle est un automate fini à deux états discrets (ON et OFF) auxquels on ajoute des informations concernant la partie continue. Ces informations sont de trois natures :

- une équation de flot dans chaque état. Il s'agit de l'équation différentielle associée à chaque état discret (par exemple $\dot{x} = 9 - x/3$ pour l'état ON) qui décrit comment la variable continue x évolue lorsque l'automate se situe dans cet état discret ;
- un invariant dans chaque état. Il s'agit de la contrainte (rouge sur la figure 4.1) associée à chaque état discret. L'automate ne pourra rester dans un état discret que tant que cet invariant sera vérifié ;
- des conditions de transition. Il s'agit de la contrainte associée à chaque transition (par exemple $x > 26$ pour la transition $switch\_off$). La transition ne pourra être effectuée que si la contrainte est vérifiée.

Exemple d'exécution. Voici, schématiquement, comment va s'exécuter ce modèle. Nous supposerons que l'automate commence dans l'état ON et que la température est de 18 ($x = 18$). Comme l'automate est dans l'état ON, que l'invariant de cet état est vérifié ($x \leq 26$) et que la condition de changement d'état ($x \geq 26$) ne l'est pas, l'automate va rester dans l'état ON. La variable x est alors modifiée selon l'équation différentielle $\dot{x} = 9 - x/3$; elle suit donc la courbe (1) de la figure 4.2, jusqu'à ce que $x = 26$. Lorsque $x = 26$, une transition continue n'est plus possible car elle invaliderait l'invariant, et la condition de saut est validée, la transition $switch\_off$ est donc exécutée. L'automate se trouve alors dans la situation suivante : $x = 26$ et l'état est OFF. La variable x est maintenant modifiée par l'équation différentielle $\dot{x} = -x/3$ jusqu'à ce que $x = 19$, c'est-à-dire qu'elle suit la courbe (2) de la figure 4.2. Lorsque $x = 19$, la transition $switch\_on$ est exécutée et l'automate se retrouve dans l'état ON, avec $x = 19$. Clairement, l'automate va alors osciller entre les états ON et OFF et la variable x va répéter le motif de la figure 4.2.

4.2 Les automates hybrides

Nous définissons maintenant formellement la notion d'automate hybride, d'exécution d'un automate hybride et de trajectoire hybride.

4.2.1 Formalisme

Soit Y un ensemble fini de variables $Y = \big(y_1, \ldots, y_n\big)$.

Définition 4.1 (Prédicat atomique) Un prédicat atomique sur les variables Y est un prédicat de la forme $f\big(y_1, \ldots, y_n\big) \odot c$, avec $f \in \mathbb{R}^n \to \mathbb{R}$, $c \in \mathbb{R}$ et $\odot \in \{<, \leq, =, \geq, >\}$.

Exemple 4.1 $x + y \leq 2$ et $y = 3$ sont deux prédicats atomiques sur $Y = \{x, y\}$.

Définition 4.2 (Prédicat) Un prédicat sur les variables Y est une conjonction positive de prédicats atomiques sur Y. Nous noterons $Pred(Y)$ l'ensemble des prédicats sur les variables Y.

Exemple 4.2 $\big(x + y \leq 2\big) \wedge \big(y = 3\big)$ est un prédicat sur $Y = \{x, y\}$.

Définition 4.3 (Valuation) Une valuation σ des variables Y est une fonction $\sigma : Y \to \mathbb{R}^n$ qui associe à chaque $y_i \in Y$ une valeur $\sigma(y_i) \in \mathbb{R}$. Nous noterons $\mathcal{V}_Y$ l'ensemble des valuations des variables Y. De plus, pour une fonction $\rho : [0, r] \to \mathcal{V}_Y$, nous noterons $\rho_y : [0, r] \to \mathbb{R}$ la fonction définie par $\forall t \in [0, r],\ \rho_y(t) = \rho(t)(y)$.

Étant donnés un prédicat $\phi \in Pred(Y)$ et une valuation σ des variables Y, nous noterons $\big[\phi\big]_\sigma$ l'évaluation de ϕ en remplaçant chaque variable y_i par $\sigma\big(y_i\big)$. Pour un prédicat ϕ nous noterons $[\![\phi]\!]$ l'ensemble des valuations qui vérifient ϕ :

$$[\![\phi]\!] = \big\{\sigma \mid \big[\phi\big]_\sigma = true\big\} \tag{4.1}$$

Définition 4.4 (Automate hybride) Un automate hybride est un 6-uplet $\big(X, S, F, L, I, T\big)$ tel que :

- X est un ensemble de variables continues $X = \big(x_1, \ldots, x_n\big)$;
- S est un ensemble d'états discrets $S = \big(s_1, \ldots, s_m\big)$;
- $F : S \to Pred(X \cup \dot{X})$ est une fonction associant à chaque état un prédicat portant sur les variables de X ainsi que sur leurs dérivées $\dot{X}$. Pour chaque état $s \in S$, $F(s)$ décrit par une relation entre $\dot{X}$ et X l'évolution des variables continues dans S ;
- L est un ensemble fini de *labels* ;
- $I : S \to Pred(X)$ est une fonction associant à chaque état un prédicat portant sur les variables X. Pour chaque état $s \in S$, $I(s)$ est un invariant qui doit être vérifié par les variables X à chaque instant où le système se trouve en l'état s ;
- $T \subseteq S \times L \times S \times Pred(X) \times Pred(X \cup X')$ décrit les transitions discrètes entre les états. Un élément $(s, l, s', \phi, \psi) \in T$ représente la transition de l'état s à l'état s', portant le label l, qui ne sera exécutée que si ϕ est vérifié et qui mettra à jour les variables continues suivant ψ. X' est l'ensemble $\big(x'_1, \ldots, x'_n\big)$ représentant les variables X après la transition. ϕ est appelé prédicat de garde et ψ prédicat de reset.

Un automate hybride génère un système de transitions que nous formaliserons dans la section 4.2.2. Intuitivement un état du système hybride comprend un état discret et une valuation pour les variables continues, et les transitions sont de deux ordres. D'une part, des transitions temporelles (ou continues) font évoluer les variables continues tout en restant dans un même état discret selon les équations de flot de cet état. D'autre part, les transitions discrètes, décrites par l'ensemble T, permettent au système de changer d'état discret : si $(s, l, s', \phi, \psi) \in T$, il y aura une transition de l'état s vers l'état s' si les variables continues satisfont ϕ, et la valeur de ces variables sera modifiée de telle sorte que ψ soit vraie. Par exemple, si $\phi = (x > 3)$ et $\psi = (x' \geq x + 1)$ et si la valuation de x est 4, alors la transition sera activée et la nouvelle valeur de x dans l'état s' sera supérieure à 5.

Par un léger abus de notation nous omettrons dans les prédicats de reset des transitions discrètes les prédicats atomiques de la forme $x'_i = x_i$ qui ne modifient pas la valeur d'une variable continue. Si aucune variable n'est modifiée, le prédicat de reset sera $true$.

Exemple 4.3 (Thermostat) L'automate hybride représentant le système du thermostat simple (figure 4.1) est encodé par le 6-uplet $\big(X_t, S_t, F_t, L_t, I_t, T_t\big)$ avec :
- $X_t = \{x\}$;
- $S_t = \{ON, OFF\}$;
- $F(ON) = \big(\dot{x} - 9 + x/3 = 0\big)$ et $F(OFF) = \big(\dot{x} + x/3 = 0\big)$;
- $L = \big\{switch\_on, switch\_off\big\}$;
- $I(ON) = x \leq 26$ et $I(OFF) = x \geq 19$;
- $T_t = \big\{\big(ON, switch\_off, OFF, x \geq 26, true\big), \big(OFF, switch\_on, ON, x \geq 19, true\big)\big\}$.

Nous adopterons les conventions suivantes pour la représentation graphique des automates hybrides. Chaque état discret s sera représenté par un cercle (ou un rectangle) comprenant son nom en rouge, son équation de flot $F(s)$ en noir et son invariant $I(s)$ en bleu. Les transitions discrètes seront représentées par des flèches entre les états. Une transition $(s, l, s', \phi, \psi) \in T$ se traduira par une flèche entre l'état s et l'état s', le label sera inscrit en bleu au milieu de la flèche, ϕ sera écrit au début et ψ à la fin de la flèche. Si $\psi = true$, nous l'omettrons.

4.2.2 Exécution d'un automate

Il existe plusieurs façons de définir l'exécution d'un automate hybride. La plus simple est de décrire la sémantique opérationnelle de l'automate comme un système de transition [Hen96] entre des états hybrides. Nous en donnons ici une version légèrement différente, compatible avec le formalisme donné dans la section 4.2.1.

Définition 4.5 (État d'un automate hybride) Soit $H = \big(X, S, F, L, I, T\big)$ un automate hybride. Un état de H est un couple (s, σ) tel que $s \in S$ est un état discret du système et $\sigma \in \mathcal{V}_X$ est une valuation des variables X de l'automate.

Remarque Parmi tous les états de l'automate, certains sont impossibles car ne vérifiant pas les invariants associé aux états discrets. Nous ne nous intéresserons donc qu'aux états dits *acceptables* qui vérifient ces invariants.

Définition 4.6 (États acceptables) Soit $H = \big(X, S, F, L, I, T\big)$ un automate hybride et (s, σ) un état de cet automate ; (s, σ) sera dit acceptable si et seulement si $\sigma \in [\![I(s)]\!]$. Nous noterons $\mathcal{S}_H$ l'ensemble des états acceptables de l'automate H.

Exemple 4.4 Pour l'automate H_t représentant le thermostat (exemple 4.3), les états $(ON, \{x \mapsto 21\})$ et $(OFF, \{x \mapsto 21\})$ sont acceptables alors que l'état $(ON, \{x \mapsto 32\})$ ne l'est pas.

Définition 4.7 (Système de transitions hybrides) Soit $H = \big(X, S, F, L, I, T\big)$ un automate hybride. H engendre un système de transitions étiquetées $LTS_H = \big(\mathcal{S}_H, \Lambda, \xrightarrow{\lambda}\big)$ tel que :

- $\mathcal{S}_H$ est l'ensemble des états acceptables de l'automate H.
- $\Lambda = \mathbb{R}_+ \cup \{l \ : \ \exists (s, l, s', \phi, \psi) \in T\}$.
- $\xrightarrow{\lambda}$ est définie par :

1. $\forall l \in L$, on a une relation $(s, \sigma) \xrightarrow{l} (s', \sigma')$ si et seulement si $(s, \sigma) \in \mathcal{S}_H$ et $(s', \sigma') \in \mathcal{S}_H$ et s'il existe $(s, l, s', \phi, \psi) \in T$ telle que $\sigma \in [\![\phi]\!]$ et la valuation $v \in \mathcal{V}_{X \cup X'}$ définie par pour $\forall x \in X,\ v(x) = \sigma(x),\ v(x') = \sigma'(x)$ vérifie $v \in [\![\psi]\!]$.
2. $\forall t \in \mathbb{R}_+$, on a une relation $(s, \sigma) \xrightarrow{t} (s, \sigma')$ si et seulement si il existe une fonction $\rho : [0, t] \to \mathcal{V}_X$ telle que :
   (a) $\forall x \in X,\ \rho_x$ est continûment dérivable ;
   (b) $\rho(0) = \sigma,\ \rho(t) = \sigma'$;
   (c) $\forall \tau \in [0, t],\ \rho(\tau) \in I(s)$ et la valuation $v \in \mathcal{V}_{X, \dot{X}}$ définie par $\forall x_i \in X,\ v(x_i) = \rho_{x_i}(\tau),\ v(\dot{x_i}) = \dot{\rho_{x_i}}(\tau)$ vérifie $v \in [\![F(l)]\!]$.

Étant donné un état initial $\chi_0 = (s_0, \sigma_0) \in \mathcal{S}_H$, une exécution de l'automate hybride H est une suite, finie ou non, de transitions $\chi_0 \xrightarrow{\lambda_0} \chi_1 \xrightarrow{\lambda_1} \dots$ telle que $\forall i \in \mathbb{N},\ \chi_i \in \mathcal{S}_H$ et $(\chi_i, \xrightarrow{\lambda_i}, \chi_{i+1}) \in LTS_H$.

Exemple 4.5 (Exécution du thermostat) L'exécution de l'automate hybride H_t représentant le système du thermostat génère l'exécution suivante pour un état initial $(ON, \{x \mapsto 18\})$:

$$
\begin{aligned}
(ON, \{x \mapsto 18\}) &\xrightarrow{t_0} (ON, \{x \mapsto 26\}) \\
&\xrightarrow{switch\_off} (OFF, \{x \mapsto 26\}) \\
&\xrightarrow{t_1} (OFF, \{x \mapsto 19\}) \\
&\xrightarrow{switch\_on} (ON, \{x \mapsto 19\}) \\
&\xrightarrow{t_2} (ON, \{x \mapsto 26\}) \\
&\xrightarrow{switch\_off} \dots
\end{aligned}
$$

L'exécution d'un automate à partir d'un état initial représente donc tous les changements d'états (discrets ou continus) de cet automate au cours du temps. Cette suite de changements peut être unique ou non, finie ou infinie. En effet, on voit bien que dans certains états, l'exécution d'un automate peut se bloquer car l'évolution continue des variables définies par l'équation différentielle mène les variables hors de l'invariant associé à l'état discret. À l'inverse, comme les transitions discrètes sont supposées être immédiates, on voit bien qu'il est possible dans certains cas d'en effectuer une infinité en un temps fini. Ce phénomène, clairement physiquement impossible, est connu sous le nom d'effet Zénon [ZJLS01]. Dans la suite, nous définissons formellement les notions d'exécutions infinies et Zénon, puis nous donnerons (section 4.2.3) des caractérisations des automates acceptant de telles exécutions.

Définition 4.8 (Durée d'une exécution) Soit $H = \big(X, S, F, L, I, T\big)$ un automate hybride et $\chi = \chi_0 \xrightarrow{\lambda_0} \chi_1 \xrightarrow{\lambda_1} \dots$ une exécution de H, avec $\lambda_i \in L \cup \mathbb{R}_+$. La durée $|\chi|$ de l'exécution est

définie par :

$$|\chi| = \sum_{\lambda_i \notin L} \lambda_i \ . \tag{4.2}$$

Parmi toutes les exécutions possibles d'un automate, celle de durée maximale (si elle existe) sera appelée *exécution maximale*.

Définition 4.9 (Exécution infinie) Soit $H = \big(X, S, F, L, I, T\big)$ un automate hybride. Une exécution $\chi_0 \xrightarrow{\lambda_0} \chi_1 \xrightarrow{\lambda_1} \dots$ de H est dite infinie si la suite de transitions est infinie.

Définition 4.10 (Exécution Zénon) Soit $H = \big(X, S, F, L, I, T\big)$ un automate hybride. Une exécution $\chi = \chi_0 \xrightarrow{\lambda_0} \chi_1 \xrightarrow{\lambda_1} \dots$ de H est dite Zénon si elle est infinie et $|\chi| < \infty$.

Une exécution Zénon correspond donc à un phénomène physiquement impossible mais permis par le modèle : une infinité de transitions sont exécutées en un temps fini. Clairement, l'existence d'exécutions Zénon pour un automate donné pose des problèmes quant à la simulation et la vérification de cet automate. En effet, il est impossible, en présence d'exécution Zénon, de connaître le comportement de l'automate après un temps fini alors que le comportement du système est défini après ce délai. L'étude des comportements Zénon est donc un enjeu important de la vérification des automates hybrides. De nombreux travaux portent sur l'analyse de la présence ou non de ces comportements [HLMR05, ZJLS01], ainsi que sur la transformation de systèmes présentant des comportements Zénon en système équivalent, non Zénon [JELS99, JLSE99]

4.2.3 Cas particuliers

Nous définissons ici quelques cas particuliers d'automates hybrides qui seront important pour les sections suivantes. Nous commençons par les automates déterministes. Intuitivement, un automate est déterministe si pour un état initial donné, il n'a qu'une évolution possible. Dans notre formalisme, cela se traduit par la définition 4.11.

Définition 4.11 (Automate hybride déterministe) Soit $H = \big(X, S, F, L, I, T\big)$ un automate hybride. H est dit déterministe si :

1. $\forall (s, l_1, s_1, \phi_1, \psi_1), (s, l_2, s_2, \phi_2, \psi_2) \in T$ avec $s_1 \neq s_2$, $[\![\phi_1]\!] \cap [\![\phi_2]\!] = \emptyset$, autrement dit deux transitions discrètes vers deux états différents ne peuvent être activées simultanément ;
2. $\forall (s, l, s', \phi, \psi) \in T,\ \forall \sigma \in \mathcal{V}_X \ : \ \sigma \in [\![\phi]\!],\ \{\sigma' \in \mathcal{V}_{x'} \ : \ \sigma \cup \sigma' \in [\![\psi]\!]\}$ est un singleton, autrement dit les prédicats de resets sont déterministes ;
3. $\forall (s, \sigma) \in \mathcal{S}_H,\ \{\sigma' \in \mathcal{V}_{\dot{x}} \ : \ \sigma \cup \sigma' \in [\![F(s)]\!]\}$ est un singleton, autrement dit les prédicats de flot sont déterministes.

Théorème 4.1 ([LJS$^+$03]) *Un automate hybride est déterministe si et seulement si pour tout état initial $(s_0, \sigma_0) \in \mathcal{S}_H$, il existe une unique exécution maximale.*

Certaines exécutions d'un automate hybride peuvent être bloquées si l'automate est dans un état discret s alors que les variables continues ne vérifient pas l'invariant $I(s)$ et qu'aucune transition discrète ne peut être activée. Nous caractérisons donc l'ensemble des automates non bloquants (définition 4.12) différemment de [LJS$^+$03] : nous donnons une caractérisation fondée sur les invariants de l'automate et prouvons ensuite (théorème 4.2) le caractère non bloquant.

Définition 4.12 (Automate hybride non bloquant) Soit $H = \big(X, S, F, L, I, T\big)$ un automate hybride. H est dit non bloquant si et seulement si pour tout état $(s, \sigma) \in \mathcal{S}_H$, on a :

$$\Big(\nexists t \in \mathbb{R}_+ \ : \ (s,\sigma) \xrightarrow{t} (s,\sigma')\Big) \Rightarrow \exists (s',\sigma') \in \mathcal{S}_H,\ \exists l \in L \ : \ (s,\sigma) \xrightarrow{l} (s',\sigma') \ .$$

Théorème 4.2 ([LJS+03]) *Si l'automate hybride $H = \big(X, S, F, L, I, T\big)$ est non bloquant, alors il possède une exécution infinie pour tout état initial $(s_0, \sigma_0) \in \mathcal{S}_H$.*

Une classe importante d'automates est la classe des automates linéaires. Un prédicat sera dit *linéaire* s'il s'écrit comme la combinaison de prédicats atomiques $\phi_1, \ldots, \phi_n$ tels que chaque ϕ_i est de la forme $\sum_{i=1}^{n} \lambda_i x_i \odot c$ où c, λ_i sont des entiers et $\odot \in \{<, \leq, =, \geq, >\}$.

Définition 4.13 (Automate hybride linéaire) Un automate hybride $H = \big(X, S, F, L, I, T\big)$ est dit linéaire si et seulement si :

1. $\forall s \in S \ : \exists k_1^s, \ldots, k_n^s \in \mathbb{R}$, $F(s) = \bigwedge_{x_i \in X} \dot{x_i} = k_i^s$.
2. pour tout état $s \in S$, $I(s)$ est un prédicat linéaire sur les variables X.
3. pour toute transition $(s, l, s', \phi, \psi) \in T$, ϕ et ψ sont des prédicats linéaires sur X et $X \cup X'$, respectivement.

Nous définissons enfin les automates rectangulaires. Nous donnons encore une fois une version légèrement différente mais équivalente de [HKPV95]. Un prédicat sur les variables $X = \big(x_1, \ldots, x_n\big)$ est dit *rectangulaire* s'il s'écrit comme la conjonction de prédicats $\phi_1, \ldots, \phi_n$ tels que chaque prédicat ϕ_i est de la forme $c_i \leq x_i \wedge x_i \leq C_i$. Nous dirons que le prédicat ϕ_i définit un rectangle pour la variable x_i.

Définition 4.14 (Automate hybride rectangulaire) Un automate hybride $H = \big(X, S, F, L, I, T\big)$ est dit rectangulaire si et seulement si :

1. pour tout état $s \in S$, $I(s)$ est un prédicat rectangulaire sur les variables X.
2. pour toute transition $(s, l, s', \phi, \psi) \in T$, ϕ est un prédicat rectangulaire sur les variables X et ψ est un prédicat rectangulaire sur les variables Y' pour $Y \subseteq X$.
3. il existe un prédicat rectangulaire $flow$ sur les variables $\dot{X}$ tel que pour tout état $s \in S$, $F(s) = flow$.

Dans un automate rectangulaire, les évolutions continues de chaque état sont donc toutes régies par la même inégalité différentielle $\bigwedge_{i=1}^{n} c_i \leq \dot{x_i} \leq C_i$. De plus, pour chaque transition (s, l, s', ϕ, ψ), le prédicat de reset est de la forme $\bigwedge_{i=1}^{p} c_i \leq \dot{y_i} \leq C_i$ pour un sous ensemble Y des variables continues. Nous dirons que la transition réinitialise les variables de Y. Lorsque les prédicats de flot sont différents pour les différents états du système (mais restent rectangulaire), nous dirons que l'automate hybride est *multirectangulaire*.

Définition 4.15 (Automate hybride multirectangulaire) Un automate hybride $\big(X, S, F, L, I, T\big)$ est dit multirectangulaire s'il vérifie les conditions 1 et 2 de la définition 4.14 et que, pour tout état $s \in S$, $F(s)$ est un prédicat rectangulaire sur les variables $\dot{X}$.

4.2.4 Exemple des deux réservoirs

L'automate. Nous donnons ici l'automate hybride correspondant au système des deux réservoirs (voir section 1.6). L'automate possède 6 états (voir figure 4.1) correspondant aux situations suivantes :

- les deux vannes sont ouvertes, et $h_2 < H$ (état $OPEN$) ;
- les deux vannes sont ouvertes, et $h_2 \geq H$ (état $OPEN_b$) ;
- la vanne 1 est fermée (état $V_1CLOSED$) ;
- la vanne 2 est fermée, et $h_2 < H$ (état $V_2CLOSED$) ;
- la vanne 2 est fermée, et $h_2 \geq H$ (état $V_2CLOSED_b$) ;
- les deux vannes sont fermées.

On peut noter qu'il n'est pas nécessaire de différencier, dans le cas où la vanne 1 est fermée, les cas où h_2 est inférieur ou supérieur à H car les évolutions continues sont les mêmes. Les transitions entre ces états sont alors de 2 ordres. D'une part, des transitions sont commandées par le contrôleur lorsque les hauteurs d'eau atteignent les valeurs critiques (par exemple, la transition entre $OPEN$ et $V_1CLOSED$). D'autre part, des transitions sont imputables à la dynamique continue elle même, lorsque le niveau d'eau h_2 franchit H (c'est le cas de la transition entre $OPEN$ et $OPEN_b$).

Formellement, l'automate hybride est donné par le septuplet $H_r = \big(X_r, S_r, F_r, L_r, I_r, T_r\big)$ avec :

- $X_r = \{h_1, h_2\}$;
- $S_r = \{OPEN, OPEN_b, CLOSED, V_1CLOSED, V_2CLOSED, V_2CLOSED_b\}$;
- $F_r : \begin{cases} OPEN & \mapsto & \dot{h_1} = i - k_1\sqrt{h_1} \wedge \dot{h_2} = k_1\sqrt{h_1} - k_2\sqrt{h_2} \\ OPEN_b & \mapsto & \dot{h_1} = i - k_1\sqrt{h_1 - h_2 + H} \wedge \dot{h_2} = k_1\sqrt{h_1 - h_2 + H} - k_2\sqrt{h_2} \\ CLOSED & \mapsto & \dot{h_1} = 0 \wedge \dot{h_2} = 0 \\ V_1CLOSED & \mapsto & \dot{h_1} = i \wedge \dot{h_2} = -k_2\sqrt{h_2} \\ V_2CLOSED & \mapsto & \dot{h_1} = i - k_1\sqrt{h_1} \wedge \dot{h_2} = k_1\sqrt{h_1} \\ V_2CLOSED_b & \mapsto & \dot{h_1} = i - k_1\sqrt{h_1 - h_2 + H} \wedge \dot{h_2} = k_1\sqrt{h_1 - h_2 + H} \end{cases}$
- $L_r = \{open\_v_1, close\_v_1, open\_v_2, close\_v_2, h_2\_gt\_H, h_2\_lt\_H\}$;
- I_r et T_r donnés sur la figure 4.3

L'automate H_r n'est clairement pas linéaire, il n'est pas déterministe (selon les valeurs de h_1 et h_2, il est possible que les transitions $close\_v_1$ et $close\_v_2$ soient activées par le même état : on ne spécifie pas alors quelle vanne doit être fermée en premier) mais il est non-bloquant.

Exemple d'exécution. Nous choisissons les valeurs suivantes pour les constantes du système : $k_1 = 3$, $k_2 = 6$, $i = 1.5$, $H = 6$ $l_1 = 2.5$, $L_1 = 8$, $l_2 = 5$ et $L_2 = 9$. L'état initial est $\big(OPEN, \{h_1 \mapsto 3, h_2 \mapsto 7\}\big)$. Nous avons simulé, en utilisant Matlab/Simulink, l'exécution de cet automate en choisissant un algorithme de Runge-Kutta d'ordre 5 pour l'intégration numérique des équations différentielles (avec un pas constant de 10^{-2}). Les planches Simulink représentant l'automate hybride sont données en annexe A. Nous obtenons les courbes de la figure 4.4 pour les valeurs de h_1 et h_2.

4.3 Autres modèles de systèmes hybrides

Parallèlement aux automates hybrides, plusieurs extensions hybrides de systèmes discrets ont été proposées. Nous présentons dans cette partie plusieurs de ces modèles. Dans un premier temps, nous évoquons les modèles hybrides reposant sur une extension des calculs de processus classiques. Nous présentons ensuite l'extension du paradigme de programmation à contraintes concurrentes (*Concurrent Constraint Programming* [SRP91]) pour le calcul hybride.

4.3.1 Calcul de processus hybrides

De nombreux modèles de calcul de processus hybrides ont été proposés. Il existe des extensions hybrides des *Communicating Sequential Processes* (CSP [Hoa78, Hoa85]) de Hoare (Timed-CSP entre autres), des *Algebra of Communicating Processes* (ACP [BW90]) et du π-calcul de Milner [Mil99]. Nous présentons ici deux de ces extensions en insistant sur les spécificités de chacune. Pour

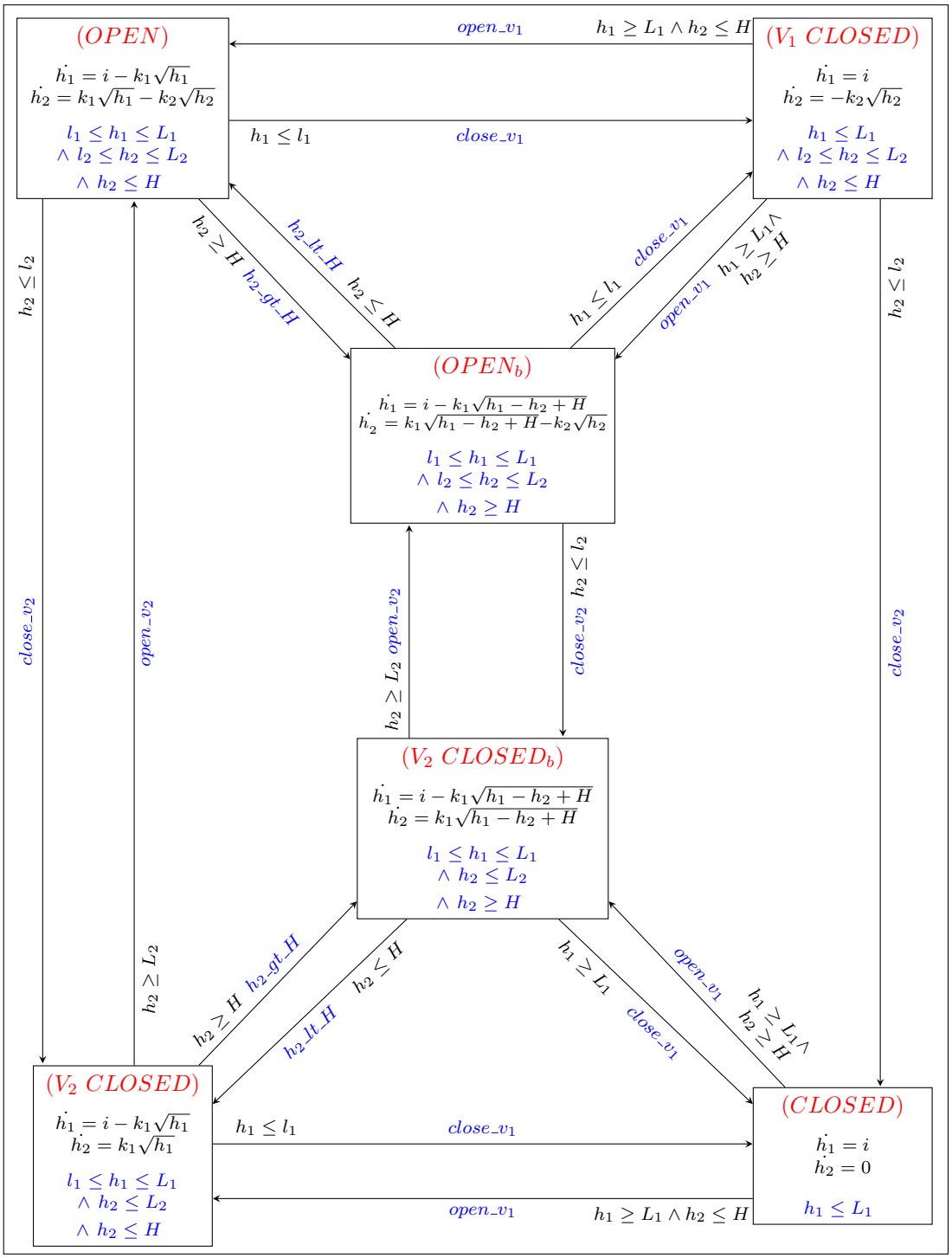


FIG. 4.3 – Automate hybride représentant le système des deux réservoirs. Remarquons que nous supposons les fermetures et ouvertures des vannes instantanées.

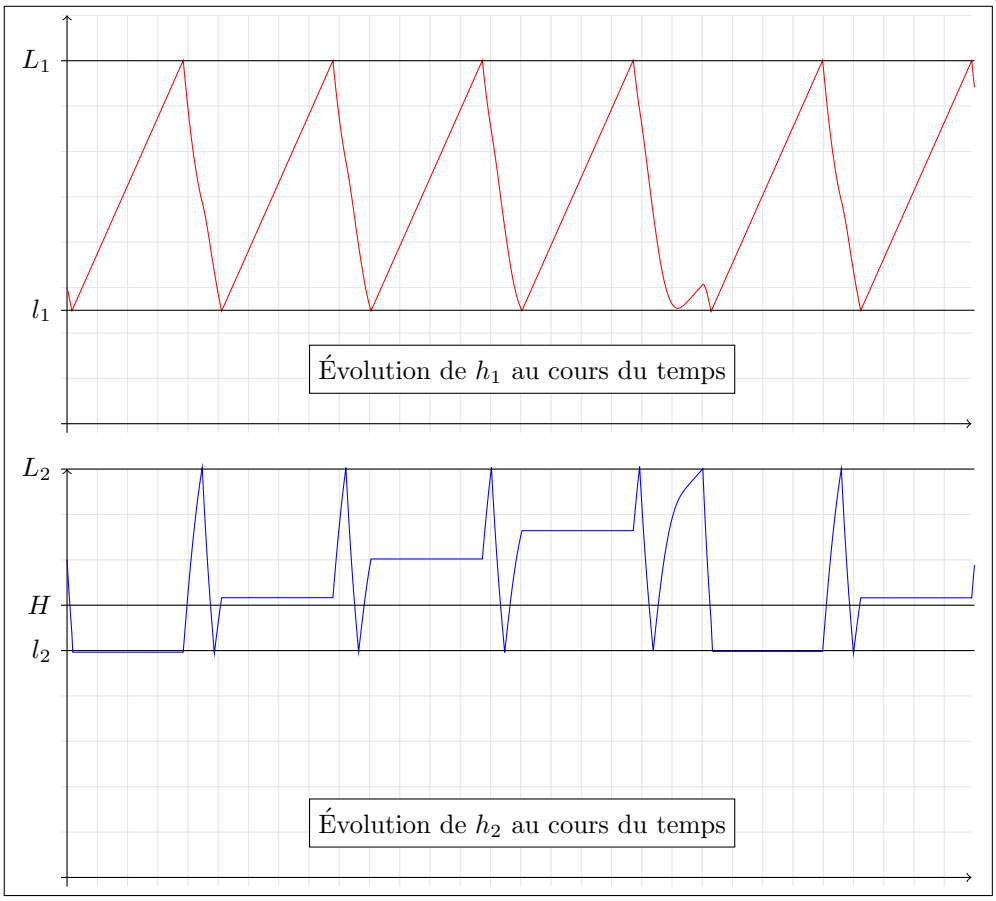


FIG. 4.4 – Variation des hauteurs d'eau dans chacun des deux réservoirs lors d'une simulation d'une exécution de l'automate de la figure 4.3.

chaque calcul de processus, nous donnons un aperçu rapide de sa syntaxe et de sa sémantique, puis nous expliquons comment l'exemple des deux réservoirs peut s'encoder dans ce formalisme.

ϕ-Calculus

Le ϕ-calcul [RS02, RS03] est une extension du π-calcul de Milner pour modéliser des systèmes hybrides concurrents et reconfigurables. L'ajout de phénomènes hybrides se fait via un processus qui représente l'environnement et des actions qui permettent de le manipuler. Un système hybride est donc un couple (E, P) tel que E représente l'environnement et P est un processus du ϕ-calcul.

Un environnement E est un triplet $(s; F; I)$ tel que s représente l'état courant de l'environnement (c'est une valuation des variables continues), F représente les équations de flot (F associe à chaque variable continue une équation différentielle) et I représente l'invariant (I est un prédicat sur les variables continues). Ainsi, l'environnement $(x = 0; \dot{x} = 1; x \leq 3)$ est l'environnement qui est initialement à $x = 0$ et qui peut évoluer selon l'équation différentielle $\dot{x} = 1$ tant que $x \leq 3$, c'est-à-dire qu'il peut évoluer pendant 3 secondes.

Un processus du ϕ-calcul est un processus du π-calcul avec les actions supplémentaires suivantes :

- une action de changement d'état continu de la forme $c \doteq d$ qui met à jour instantanément les variables continues ; par exemple, l'action $x \doteq 3$ donnera à x la valeur 3.
- une action de changement d'équation différentielles de la forme $F \doteq G$ qui change instantanément les équations de flot.

– une action de changement d'invariants de la forme $I \doteq J$ qui change instantanément l'invariant associé à l'état continu.

Une action e sur l'environnement est donc encodée par $e = \Psi \rightarrow \left(c \doteq d; F \doteq G; I \doteq J\right)$ et elle est considérée comme une action classique du π-calcul pour ce qui concerne les règles de composition (somme et parallélisme). Ψ est un prédicat tel que la transition sera activée si et seulement si Ψ est vérifié. Les transitions sur l'environnement permettent aux processus discrets de *créer* arbitrairement un nouvel environnement continu. En cela, le ϕ-calcul est bien une extension du π-calcul où de nouveaux processus discrets peuvent être crées dynamiquement.

Un système hybride (E, P) peut alors évoluer de trois façons. Les actions purement discrètes du π-calcul ne changent que le processus P. Les actions sur l'environnement peuvent être effectuées pour mettre à jour E, et dans ce cas à la fois P et E sont modifiées. Ces deux types d'actions sont supposées immédiates. Le système peut également évoluer selon des transitions de flot qui font changer l'environnement continûment d'après les équations différentielles choisies. Une transition de flot ne peut s'effectuer que tant qu'aucune transition sur l'environnement n'est activée. Ces dernières représentent en effet les changements de mode continu, et ont donc priorité sur les transitions de flot qui décrivent l'évolution du système dans un mode donné.

$$S = (\emptyset, [TRUE \rightarrow \begin{pmatrix} h_1 \doteq 3 \\ h_2 \doteq 7 \end{pmatrix}; F \doteq \begin{pmatrix} \dot{h_1} : i - k_1\sqrt{h_1 - h_2 + H} \\ \dot{h_2} : k_1\sqrt{h_1 - h_2 + H} - k_2\sqrt{h_2} \end{pmatrix}; I \doteq \begin{pmatrix} l_1 \leq h_1 \leq L_1 \\ l_2 \leq h_2 \leq L_2 \\ h_2 \geq H \end{pmatrix}].OPEN_b)$$

$$\begin{aligned}
OPEN_b &= (h_1 \leq l_1) \rightarrow [\emptyset; F \doteq \begin{pmatrix} \dot{h_1} : i \\ \dot{h_2} : -k_2\sqrt{h_2} \end{pmatrix}; I \doteq \begin{pmatrix} h_1 \leq L_1 \\ l_2 \leq h_2 \leq L_2 \\ h_2 \leq H \end{pmatrix}].V_1CLOSED \\
&+ (h_2 \leq H) \rightarrow [\emptyset; F \doteq \begin{pmatrix} \dot{h_1} : i - k_1\sqrt{h_1} \\ \dot{h_2} : k_1\sqrt{h_1} - k_2\sqrt{h_2} \end{pmatrix}; I \doteq \begin{pmatrix} l_1 \leq h_1 \leq L_1 \\ l_2 \leq h_2 \leq L_2 \\ h_2 \leq H \end{pmatrix}].OPEN \\
&+ (h_2 \leq l_2) \rightarrow [\emptyset; F \doteq \begin{pmatrix} \dot{h_1} : i - k_1\sqrt{h_1 - h_2 + H} \\ \dot{h_2} : k_1\sqrt{h_1 - h_2 + H} \end{pmatrix}; I \doteq \begin{pmatrix} l_1 \leq h_1 \leq L_1 \\ h_2 \leq L_2 \\ h_2 \geq H \end{pmatrix}].V_2CLOSED
\end{aligned}$$

$$\begin{aligned}
V_1CLOSED &= \begin{pmatrix} h_1 \geq L_1 \\ h_2 \geq H \end{pmatrix} \rightarrow [\emptyset; F \doteq \begin{pmatrix} \dot{h_1} : i - k_1\sqrt{h_1 - h_2 + H} \\ \dot{h_2} : k_1\sqrt{h_1 - h_2 + H} - k_2\sqrt{h_2} \end{pmatrix}; I \doteq \begin{pmatrix} l_1 \leq h_1 \leq L_1 \\ l_2 \leq h_2 \leq L_2 \\ h_2 \geq H \end{pmatrix}].OPEN_b \\
&+ (h_2 \leq l_2) \rightarrow [\emptyset; F \doteq \begin{pmatrix} \dot{h_1} : i \\ \dot{h_2} : 0 \end{pmatrix}; I \doteq \begin{pmatrix} h_1 \leq L_1 \end{pmatrix}].CLOSED \\
&+ \begin{pmatrix} h_1 \geq L_1 \\ h_2 \leq H \end{pmatrix} \rightarrow [\emptyset; F \doteq \begin{pmatrix} \dot{h_1} : i - k_1\sqrt{h_1} \\ \dot{h_2} : k_1\sqrt{h_1} - k_2\sqrt{h_2} \end{pmatrix}; I \doteq \begin{pmatrix} l_1 \leq h_1 \leq L_1 \\ l_2 \leq h_2 \leq L_2 \\ h_2 \leq H \end{pmatrix}].OPEN
\end{aligned}$$

FIG. 4.5 – Encodage d'une partie du problème des deux réservoirs dans le ϕ-calcul. La traduction de l'automate hybride de la figure 4.3 en ϕ-calcul est immédiate.

Hybrid-χ

Le langage Hybrid-χ [vBMR+06] est un langage de processus pour modéliser les systèmes hybrides qui veut unifier les approches "systèmes dynamiques" et "informatique". Le langage intègre en effet les formalismes des deux communautés (équations différentielles discontinues, PWA mais aussi actions discrètes) et permet la mise en parallèle intuitive de processus discrets et continus. Les principales caractéristiques du langage sont :

– une caractérisation claire des variables du système en variables continues et discrètes. Parmi les variables continues, on distingue les variables algébriques qui peuvent être modifiées par une fonction non continue, et les variables de flot dont l'évolution est régie par une équation

différentielle (éventuellement discontinue). Les variables discrètes sont elles aussi séparées en variables discrètes constantes et non-constantes.
- une composition parallèle des processus avec une communication via des variables partagées ou par une synchronisation sur un canal.

Le comportement d'un processus χ (c'est-à-dire un processus du langage Hybrid-χ) est défini par un système de transitions qui contient des transitions discrètes, appelées actions, et des transitions continues appelées délais. Les actions peuvent changer instantanément la valeurs des variables discrètes non-constantes (dites *jumping variables*) et des variables algébriques. Les délais font avancer le temps, et ainsi évoluer les variables continues. Les relations entre ce langage et les autres modèles hybrides ont été très étudiées. Ainsi, il est possible d'encoder les automates hybrides en χ et une partie bien identifiée de χ est représentable par des automates hybrides. De même, il est aisé de traduire des systèmes PWA en χ.

Nous présentons la syntaxe du langage par l'exemple des deux réservoirs (figure 4.6). Les deux premières lignes correspondent aux déclarations et initialisation des variables du systèmes. Ici, v_1 et v_2 représentent l'état des vannes 1 et 2 ($v_1 = 1$ si et seulement si la vanne i est ouverte), Q_{in} représente l'écoulement de l'eau entre les deux réservoirs et Q_{out} l'écoulement par le bas du réservoir 2. Notons la variable discrète supplémentaire $h_2\_gt\_H$ qui vaudra 1 si et seulement si $h_2 \geq H$. Le processus lui même est la mise en parallèle (via l'opérateur $||$) d'un processus continu E est d'un contrôleur discret P. Le processus E est un encodage très classique des équations différentielles du système des deux réservoirs (figure 1.5) en utilisant les paramètres booléens v_1, v_2 et $h_2\_gt\_H$ pour choisir parmi les différents modes. Les processus P est encodé comme la répétition infinie (représentée par le symbole $*$ au début) d'un processus qui en fonction des évènements possibles (h_1 trop grand, h_2 trop petit, h_2 passe au dessus de H, etc.) agit sur les paramètres booléens pour choisir le bon mode continu.

$$
\begin{array}{rcl}
S & = & \langle \text{disc } v_1, v_2, H, h_2\_gt\_H, \text{ cont } h_1, h_2, \text{ alg } Q_{in}, Q_{out}, \\
 & & v_1 = 0, v_2 = 0, h_1 = 3, h_2 = 7, H = 6 \\
 & & :: E||P \rangle \\
E & = & (\dot{h_1} = Q_{in} - Q_{out_1}, \dot{h_2} = Q_{out_1} - Q_{out_2}, \\
 & & Q_{in} = i, \\
 & & Q_{out_1} = v_1 * k_1 * (h_2\_lt\_H \times \sqrt{h_1} - (1 - h_2\_lt\_H) \times \sqrt{H + h_1 - h_2})), \\
 & & Q_{out_2} = v_2 * k_2 * \sqrt{h_2}) \\
P & = & *(h_1 \leq 2.5 \rightarrow v_1 := 0; h_1 \geq 8 \rightarrow v_1 := 1; \\
 & & h_2 \geq H \rightarrow h_2\_lt\_H := 0; h_2 \leq H \rightarrow h_2\_lt\_H := 1; \\
 & & h_2 \leq 5 \rightarrow v_2 := 0; h_2 \geq 9 \rightarrow v_2 := 1;)
\end{array}
$$

FIG. 4.6 – Encodage du problème des deux réservoirs en Hybrid-χ.

4.3.2 Hybrid-CC

Hybrid-CC [GJSB95, GJS98, GSS95] est le premier formalisme permettant de modéliser des systèmes présentant un comportement continu et discret. Il s'agit d'une extension du langage de programmation par contraintes concurrents CC pour les systèmes hybrides. Un programme CC consiste en un ensemble d'agents qui s'exécutent simultanément et interagissent via un ensemble partagé de contraintes appelé *store*. Ce store contient à chaque instant une conjonction de contraintes que doivent vérifier les variables du système. Les agents disposent de deux prédicats pour interagir avec le store : *tell* qui permet d'ajouter de l'information au store, et *ask* qui interroge le store sur la validité d'une formule. Notons que l'action *ask* est bloquante : si on ne peut décider de la validité de la formule, le processus est gelé jusqu'à ce que de l'information soit ajoutée au store pour lever l'indécision. Ce mécanisme permet donc la synchronisation des agents

parallèles. L'expressivité des programmes CC est très liée au système de contraintes sous-jacent qui définit l'ensemble des prédicats disponibles pour le store. Dans le cas de Hybrid-CC, ce système de contraintes est enrichi de contraintes continues de la forme `X=0,hence dot(X)=1` qui signifie que la variable `X` suit une évolution continue donnée par l'équation différentielle $\dot{x} = 1, x(0) = 0$. L'ajout principal est donc le mot clé `hence` qui signifie que la contrainte suivante est vraie pour tous les instants futurs.

Pour permettre de calculer et simuler les programmes écrits en Hybrid-CC, plusieurs restrictions sur les contraintes `hence` et `dot` sont imposées. La première condition est une condition de stabilité qui impose que pour toutes contraintes a et b, il existe un voisinage de 0 tel que $a \Rightarrow b$ ou $a \Rightarrow \neg b$ à chaque instant du voisinage. Ainsi, on ne pourra pas écrire un agent du type `T=0, hence dot(T)=1, hence if rational(T) then A` qui supposerait l'exécution de `A` à chaque instant `T` tel que `T` est un nombre rationnel. La deuxième contrainte imposée par le langage limite l'expressivité des contraintes continues : seules les contraintes du type `dot(X,m)=r` sont acceptées, et elles signifient que la dérivée d'ordre `m` de `X` vaut `r`, où `r` est un nombre réel. Cette contrainte sur les évolutions continues permet de décider de la validité des contraintes où `X` intervient car on peut calculer exactement la fonction d'évolution de `X` au cours du temps. Le langage Hybrid-CC est donc la première formalisation des phénomènes hybrides, mais son utilité reste limitée car il n'accepte que des comportements continus triviaux. On ne pourra ainsi pas encoder le problème des deux réservoirs.

4.4 Vérification des systèmes hybrides

Les outils de modélisation que nous venons de décrire servent généralement à formaliser des systèmes hybrides dans le but d'étudier leur comportement. Des techniques de vérification pour les automates hybrides ont été développées très tôt [ACHH92, ACH+95] et pour la plupart des autres modèles, la vérification se fait via une traduction en automates hybrides et une utilisation des outils existant. Nous nous contenterons donc d'étudier le problème de la vérification des automates hybrides.

4.4.1 Problème de l'accessibilité pour les automates hybrides

La notion d'état accessible est le concept central pour la vérification des automates hybrides. Intuitivement, un état σ sera accessible depuis un ensemble d'états initiaux si et seulement si il existe une exécution de l'automate partant d'un des états initiaux et qui passe par σ.

Définition 4.16 (Relation d'accessibilité) Soit $H = \big(X, S, F, L, I, T\big)$ un automate hybride, et $(\sigma, \sigma') \in \mathcal{S}_H$ deux états acceptables. L'état σ' est dit accessible depuis σ, noté $\sigma \rightarrow^* \sigma'$, si et seulement si il existe une suite finie de transitions $\lambda_1, \ldots, \lambda_n$ telles que :

$$\sigma \xrightarrow{\lambda_1} \ldots \xrightarrow{\lambda_n} \sigma' \ .$$

Exemple 4.6 Dans le cas de l'automate hybride de la figure 4.3, l'état $\big(V_2CLOSED, \{h_1 \mapsto L_1; h_2 \mapsto H-1\}\big)$ est accessible depuis l'état $\big(\big(V_2CLOSED_b, \{h_1 \mapsto l_1; h_2 \mapsto H-1\}\big)$ par la suite de transitions suivantes, avec $\tau = \frac{L_1 - l_1}{i}$:

$$\begin{aligned}
\big(V_2CLOSED_b, \{h_1 \mapsto l_1; h_2 \mapsto H-1\}\big) &\xrightarrow{close_v_1} \big(CLOSED, \{h_1 \mapsto l_1; h_2 \mapsto H-1\}\big) \\
&\xrightarrow{\tau} \big(CLOSED, \{h_1 \mapsto L_1; h_2 \mapsto H-1\}\big) \\
&\xrightarrow{open_v_1} \big(V_2CLOSED, \{h_1 \mapsto L_1; h_2 \mapsto H-1\}\big)
\end{aligned}$$

Définition 4.17 (Ensemble accessible) Soit $H = \big(X, S, F, L, I, T\big)$ un automate hybride, et $\mathcal{I} \subseteq \mathcal{S}_H$ un ensemble initial d'états acceptables. L'ensemble des états accessibles depuis $\mathcal{I}$, noté $R\big(\mathcal{I}\big)$, est donné par :

$$R\big(\mathcal{I}\big) = \{\sigma' \in \mathcal{S}_H \ : \ \exists \sigma \in \mathcal{I}, \ \sigma \rightarrow^* \sigma'\} \tag{4.3}$$

Le problème du calcul de l'ensemble accessible [HR98] occupe une place centrale dans la vérification des propriétés de sûreté ou de vivacité des automates hybrides. En effet, pour prouver la sûreté d'un automate, on lui rajoute généralement des états dits mauvais qui représentent les exécutions du système qui ne doivent pas arriver. Il suffit alors de vérifier que l'ensemble accessible de l'automate ne contient aucun de ces états pour prouver la sûreté du système. De même, les problèmes de vivacité peuvent s'exprimer comme un problème d'accessibilité. Par exemple, pour tester si le système reste infiniment dans un ensemble d'états donné $\mathcal{I}$ (autrement dit, $\mathcal{I}$ est-il un invariant du système ?), il suffit de calculer l'ensemble accessible depuis $\mathcal{I}$ et vérifier qu'il ne possède aucun état extérieur à $\mathcal{I}$. On voit donc que le problème 4.1 est le problème central pour la vérification des automates hybrides. Nous étudions dans la suite les questions de décidabilité liées à ce problème pour nous verrons (section 4.4.2) les techniques permettant de le résoudre dans certains cas.

Problème 4.1 (Accessibilité d'un état) *Étant donnés un automate hybride H, un ensemble d'états acceptables $\mathcal{I} \subseteq \mathcal{S}_H$ et un état $\sigma \in \mathcal{S}_H$, a-t-on $\sigma \in R\big(\mathcal{I}\big)$?*

Résultats d'indécidabilité pour les différents types d'automates Le problème 4.1 est malheureusement indécidable dans le cas général. Comme le montrent les théorèmes 4.3 et 4.4, cela reste vrai même pour des dynamiques très simples. Nous ne donnons ici que quelques résultats concernant la décidabilité du problème 4.1, un panorama complet avec des preuves détaillés peut être lu dans [HKPV95, Hen96, Mil00].

Théorème 4.3 (Indécidabilité pour les automates rectangulaires [HKPV95]) *Le problème 4.1 est indécidable pour les automates hybrides rectangulaires.*

Théorème 4.4 (Indécidabilité pour les automates linéaires) *Le problème 4.1 est indécidable pour les automates hybrides linéaires.*

On voit donc que pour des classes très larges d'automates hybrides, le problème d'accessibilité est indécidable. Il existe quelques résultats de décidabilité qui concernent essentiellement les automates avec des dynamiques triviales (c'est-à-dire tels que pour tout état s, $F(s)$ est une constante positive) ou les automates *initialisés*. Un automate hybride $H = \big(X, S, F, L, I, T\big)$ est dit initialisé si et seulement si pour toute transition $(s, l, s', \phi, \psi) \in T$ et pour toute variable $v \in V$, soit le flot associé à v ne change pas après la transition $(F(s)(v) = F(s')(v))$, soit la valeur de v est modifiée par la transition. Dans le cas des automates rectangulaires, cela revient à dire que ψ a un prédicat de la forme $a \leq v' \wedge v' \leq b$ pour $a, b \in \mathbb{R}$.

Théorème 4.5 (Décidabilité pour les automates rectangulaires initialisés [HKPV95]) *Le problème 4.1 est décidable pour les automates multirectangulaires initialisés.*

4.4.2 Analyse de l'accessibilité

La validation de propriétés de sûreté pour un automate hybride se traduit naturellement en un instance du problème 4.1. Par exemple, si l'on veut montrer que le système hybride ne possède pas de comportements dangereux, il suffit de définir dans l'automate hybride des états correspondant à ces comportements dangereux et vérifier qu'ils n'appartiennent pas à l'ensemble des états accessibles. Dans la suite, un ensemble d'états sera appelée une *région*. Nous présentons ici la technique classique pour calculer la région accessible depuis une région initiale, puis nous verrons

que cette région est calculable dans le cas des automates hybrides linéaires, et nous expliquerons les techniques utilisées dans le cas des automates non-linéaires.

Soit R une région d'un automate hybride H (c'est-à-dire un ensemble d'états). Nous noterons $R^{\nearrow}$ la région accessible depuis R en laissant avancer le temps :

$$R^{\nearrow} = \left\{ \sigma \in \mathcal{S}_H \ : \ \exists \sigma' \in R, \exists t \in \mathbb{R}_+, \ \sigma' \xrightarrow{t} \sigma \right\} . \tag{4.4}$$

Par ailleurs, nous noterons $R \xrightarrow{\lambda}$ la région (potentiellement vide) accessible depuis R en effectuant la transition portant le label λ :

$$R \xrightarrow{\lambda} = \left\{ \sigma \in \mathcal{S}_H \ : \ \exists \sigma' \in R, \ \sigma' \xrightarrow{\lambda} \sigma \right\} . \tag{4.5}$$

En utilisant les définitions (4.4) et (4.5), on peut maintenant définir pour tout état discret s de l'automate la région $Reach_s(R_0)$, qui est l'ensemble des états de la forme (s, v) accessible depuis une région initiale R_0 :

$$Reach_s(R_0) = \left(\left(R_0 \cap [\![I(s)]\!]\right) \cup \bigcup_{(s',\lambda,s,\phi,\psi)\in T} Reach_{s'}(R_0) \xrightarrow{\lambda} \right)^{\nearrow} . \tag{4.6}$$

L'équation (4.6) définit donc un système d'équations récursives qui peut se comprendre ainsi. Les états (s, v) accessibles depuis R_0 sont de trois ordres :

1. soit il s'agit des états déjà présents dans R_0 : il s'agit de $R_0 \cap [\![I(s)]\!]$;
2. soit ce sont des états obtenus par une transition discrète $(s', \lambda, s, \phi, \psi)$ depuis un état (s', v') accessible depuis R_0, il s'agit donc de $\bigcup_{(s',\lambda,s,\phi,\psi)\in T} Reach_{s'}(R_0) \xrightarrow{\lambda}$;
3. soit ce sont des états obtenus en laissant avancer le temps depuis un autre état (s, v') accessible depuis R_0. C'est ce qui explique la présence de l'opérateur $\nearrow$ dans l'équation (4.6).

On définit donc l'ensemble des états accessibles comme le plus petit point fixe du système d'équations récursives (4.6), on peut donc l'appeler la sémantique collectrice de l'automate hybride par analogie à la sémantique collectrice des langages introduite au chapitre 2. La difficulté dans l'analyse des automates hybrides consiste à calculer ce point fixe. On peut comme au chapitre 2 le calculer par une itération de Kleene, mais cela nécessite le calcul explicite de l'opérateur $\nearrow$ qui est dans le cas général non calculable et reste le point délicat pour la vérification des automates hybrides [PC07]. Nous détaillons dans la suite le cas des automates linéaires pour lequel des techniques de calcul efficace existent [ACH+95, HPR94].

Remarque (1) Nous avons présenté ici l'analyse des automates *en avant*, c'est-à-dire que l'on calcule à partir d'une région initiale la région accessible. Un autre possibilité consiste à effectuer une analyse *en arrière* qui calcule, à partir de la région interdite, l'ensemble des états qui peuvent mener à cette région. On vérifie alors que la région initiale ne contient aucun de ces états. Certains outils d'analyse utilisent une combinaison de ces deux techniques pour améliorer le temps de calcul.

Remarque (2) Le calcul du plus petit point fixe du système d'équations (4.6) est analogue au calcul du plus petit point fixe nécessaire à la définition de la sémantique dénotationnelle des langages de programmations (voir la section 2.2). En fait, on peut définir la sémantique d'un automate hybride comme étant la fonction associant à un ensemble initial l'ensemble des états atteignables. Le parallèle avec une sémantique dénotationnelle des langages de programmation est alors très fort. Les techniques d'interprétation abstraite qui ont fait leurs preuves pour la vérification des logiciels peuvent alors être appliquées à la vérification des automates hybrides : l'ensemble des états initiaux est surapproximé (souvent par un polyhèdre, [CH78]) et l'opérateur $\nearrow$ est surapproximé en un opérateur abstrait. Ces techniques ne sont cependant efficaces que dans le cas des automates hybrides linéaires, que nous détaillons dans la suite. Comme nous le verrons, un opérateur de widening est également définissable pour les automates linéaires.

Cas des automates linéaires

Soit $H = \big(X, S, F, L, I, T\big)$ un automate linéaire. Les invariants associés à chaque état sont des formes linéaires et pour chaque état $s \in S$, on peut donc représenter $R_0 \cap [\![I(s)]\!]$ par un polyèdres [CH78]. De plus, les prédicats de garde et de reset sont des formes linéaires, leur action sur un polyèdres est donc linéaire et on peut alors calculer le résultat sous forme d'un polyèdres. On peut donc calculer efficacement $R \xrightarrow{\lambda}$ pour toute région polyédrique R et transition λ. Enfin, les équations différentielles sont elles-mêmes linéaires, on peut alors calculer un système de générateurs pour $R^{\nearrow}$ à partir du système de générateurs pour R [HPR94]. En utilisant les polyèdres et les opérateurs associés (notamment l'union), on peut donc calculer au moins une surapproximation des itérées de Kleene et donc une surapproximation des états accessibles. Cependant, ce calcul est potentiellement infini car l'ensemble des états du système est infini. Pour accélérer le calcul, on peut utiliser les techniques d'élargissement décrites au chapitre 2 [HPR94, HH95]. L'utilisation du domaine des polyèdres fournit un opérateur d'élargissement efficace et on peut donc calculer en un temps fini une surapproximation de tous les états accessibles. Si on peut montrer que cette surapproximation ne contient aucun des états dangereux, on sait a fortiori que ces états ne sont pas accessibles et donc que le système est sûr.

Les automates hybrides linéaires représentent la plus grande classe d'automates pour lesquels il existe un calcul efficace de l'ensemble des états accessibles. Ils jouent donc un rôle très important dans l'analyse des autres types de systèmes hybrides.

Cas des automates non linéaires

Pour les automates non linéaires [HHWT98], les opérateurs de la formule (4.6) ne sont pas calculables et on ne sait pas représenter efficacement les régions de l'automate. En particulier, l'opérateur $^{\nearrow}$ d'avancement du temps n'est pas calculable [PC07]. Plusieurs solutions ont alors été proposées pour calculer (ou au moins surapproximer) les états accessibles. Dans [HHWT98], Henzinger et al. proposent de surapproximer chaque contrainte non linéaire par une (ou plusieurs) contrainte linéaire. Ainsi, on encadre l'évolution continue par deux évolutions linéaires que l'on sait calculer. Un autre solution consiste à remplacer l'évolution continue non linéaire par un polynôme pour lequel on sait calculer une erreur uniforme [PC07], c'est-à-dire que l'on connaît une distance ϵ entre ce polynôme et la solution de l'équation différentielle non linéaire. On dispose alors d'une approximation des états accessibles et on connaît la distance entre cette approximation et les vrais états accessibles. Cette technique est toutefois difficile à mettre en place si les régions ne sont pas triviales (c'est-à-dire contenant plus d'un état). Enfin, une troisième technique consiste à utiliser les techniques d'intégration garantie présentées au chapitre 3 pour surapproximer les dynamiques continues [HHMWT00]. On peut ainsi surapproximer l'opérateur $^{\nearrow}$ en utilisant une arithmétique d'intervalle. Cette technique a été implémentée dans l'outil HyperTech et semble la plus adapté dans le cas de dynamiques fortement non linéaires.

4.4.3 Outils

HyTech

HyTech[1] [HHWT97] est le premier outil automatique utilisé pour la vérification des systèmes hybrides. Il repose sur un calcul d'accessibilité pour des automates hybrides linéaires et permet d'écrire les critères de sûreté dans un langage de programmation assez avancé. Par exemple, on peut calculer le temps total passé dans l'état ON de l'exemple 4.3 pendant une certaine durée d'exécution. On peut ainsi prouver que le système répond à des spécifications d'assez haut niveau (par exemple, le radiateur ne doit pas être allumé plus de la moitié du temps). Le logiciel peut également générer des diagnostiques si il prouve que le système n'est pas sûr : il donne un exemple d'exécution terminant dans un état dangereux.

[1] `http ://embedded.eecs.berkeley.edu/research/hytech/`

L'analyse d'accessibilité pour les automates linéaires implémentée dans HyTech est très proche de celle décrite à la section 4.4.2, les régions accessibles étant décrites par des ensembles de polyèdres convexes. Comme le problème d'accessibilité est indécidable pour la classe des automates hybrides linéaires (théorème 4.4), la procédure peut clairement ne jamais terminer. Un des défauts de HyTech est sa gestion des polyèdres : HyTech utilise des nombres entiers codés sur un nombre fini de bits (le type `int` de C) pour représenter les coefficients des polyèdres, ce qui conduit souvent à des problèmes de dépassement de capacité pour des systèmes complexes.

Le logiciel HyperTech est le successeur d'HyTech et intègre des méthodes d'intégrations garanties.

PHAVer

PHAVer[2] [Fre05] est un outil de vérification des automates hybrides plus récent, développé par Goran Frehse, et qui vise à combler les lacunes de HyTech. Le problème des dépassements de capacité est réglé par l'utilisation de la Parma Polyhedra Library [BRZH02] qui permet un calcul exact sur les polyèdres grâce à une arithmétique des entiers multi-précision. Le but de PHAVer est également de permettre l'analyse de systèmes plus complexes que ceux pris en charge par HyTech. Pour cela, PHAVer propose un algorithme d'abstraction automatique de dynamiques affines par morceaux en des dynamiques linéaires. Un partitionnement automatique des états continus permet alors de conserver une précision suffisante.

Les automates acceptés par PHAVer sont donc des automates hybrides affines avec entrées/sorties (*Hybrid I/O-Automata*, [LSVW96]), ce qui permet une analyse modulaire des systèmes hybrides de grande taille. L'analyse d'accessibilité utilisée dans PHAVer est une analyse classique étendue par un partitionnement automatique des états continus qui permet une paramétrisation par l'utilisateur de la précision de l'analyse. Des approximations sont également effectuées en cours d'analyse pour limiter l'empreinte mémoire des polyèdres calculés. PHAVer a été utilisé pour la vérification d'une version simple du système des deux réservoirs [MK06].

[2] `http ://www-verimag.imag.fr/∼frehse/phaver_web/`

Deuxième partie

Concret

Dans cette partie, nous présentons un nouveau modèle pour les systèmes hybrides qui se situe à un niveau plus bas que les automates hybrides (en bas du cycle en V). Nous présentons dans un premier temps ce modèle (section 5.2) qui ajoute à un langage de programmation impératif des constructeurs hybrides, puis nous définissons une sémantique dénotationnelle pour ce modèle. Cette sémantique est définie en trois étapes : nous définissons d'abord une sémantique pour la partie continue du système (section 5.3), puis pour la partie discrète (section 5.4) et enfin pour le système hybride dans son ensemble (section 5.5).

CHAPITRE 5

Un modèle et une sémantique pour les programmes hybrides.

Dans ce chapitre, nous introduisons un nouveau modèle de systèmes hybrides conçu pour la vérification des logiciels critiques embarqués. De la même façon que les automates hybrides sont une extension des automates à états finis pour le calcul hybride, notre modèle est une extension des langages de programmation impératifs qui introduit un comportement hybride en prenant en compte l'environnement extérieur dans lequel le programme est exécuté. Dans un premier temps (section 5.1), nous expliquons les raisons qui nous ont amenés à définir ce nouveau modèle et les caractéristiques des programmes critiques embarqués qui ont influé dans nos choix. Ensuite, nous présentons ce modèle en donnant sa syntaxe (section 5.2) puis sa sémantique (sections 5.3 à 5.5). Nous terminons par la modélisation d'un système dérivé du système des deux réservoirs et le calcul de sa sémantique (section 5.6).

5.1 Du programme discret au programme hybride

5.1.1 Caractéristiques hybrides des programmes embarqués

Comme nous l'avons dit en introduction, les programmes critiques embarqués interagissent avec l'environnement physique dans lequel ils sont exécutés. Leur exécution est purement discrète, alors que l'évolution de l'environnement extérieur est continue : un programme embarqué doit donc être vu comme une composante d'un système hybride. Cependant, certaines caractéristiques intrinsèques de ces programmes rendent les modèles que nous avons présentés au chapitre 4 peu adaptés pour représenter et analyser ce type de systèmes hybrides. Cette section vise à expliquer ce constat.

D'un point de vue très général, un système hybride consiste en deux processus de natures différentes s'exécutant en parallèle : la partie discrète et la partie continue. Ces deux processus communiquent à travers des interfaces : les capteurs pour la communication du processus continu vers le processus discret, les actionneurs pour la communication en sens inverse (voir la figure 1.4). Selon les modèles utilisés pour représenter chaque processus et le choix du type de communication pour modéliser les capteurs et les actionneurs (communication synchrone ou asynchrone, communication par envoi de messages ou via une mémoire partagée), on obtient des modèles hybrides ayant des comportements très différents. La principale difficulté dans la modélisation d'un système hybride vient de la présence d'actionneurs : ces derniers créent une dépendance cyclique entre l'évolution discrète et l'évolution continue ce qui rend la dynamique du système hybride complexe à définir. Une contribution de cette thèse est de prendre en compte ces actionneurs et donc la rétroaction entre le programme et son environnement. Remarquons que contrairement à

de nombreux modèles hybrides, nous n'autoriserons pas un actionneur à modifier directement la valeur d'une variable continue ; il ne pourra que modifier sa dynamique. Ceci n'est pas le cas dans la théorie des automates hybrides par exemple, où un changement d'état discret peut s'accompagner d'un changement discontinu des valeurs des variables continues. Cette possibilité nous semble trop peu réaliste (un moteur ne peut pas changer instantanément la position de la voiture, mais uniquement son accélération) et nous l'avons donc exclue de notre modèle.

La modélisation de la partie continue d'un système hybride est souvent donnée par les lois de la physique sous forme d'équations différentielles ou d'équations aux dérivées partielles. Ce cadre très général permet d'appréhender de nombreux comportements continus, mais sa trop grande expressivité et les résultats d'indécidabilité qui en découlent (théorèmes 4.3 et 4.4) ont poussé la plupart des outils de modélisation et d'analyse à limiter le type d'équations différentielles permises ou à choisir d'autres modélisations comme les fonctions affines par morceaux (*Piecewise Affine Systems* ou PWA). Les choix effectués pour modéliser la partie continue ont en fait assez peu d'incidence sur le comportement global du système, ils ont plus une fonction limitative quand à l'expressivité du modèle. Remarquons que quelle que soit la modélisation choisie, un modèle pour la partie continue est toujours composé d'un ensemble de modèles simples représentant chacun un "mode continu", ce qui permet à la partie discrète d'influer sur l'évolution continue en en choisissant un. C'est généralement ainsi que sont modélisés les actionneurs. Dans l'exemple 4.3 du thermostat, la partie continue possède deux modes : un correspondant à l'état où le radiateur est allumé et un correspondant à l'état où le radiateur est éteint.

La théorie des automates hybrides modélise la partie discrète du système hybride par un automate fini et associe à chaque état de l'automate un mode continu. Le système hybride est donc construit comme le produit cartésien d'un automate fini et de l'ensemble des modes continus. Les actionneurs sont alors représentés par des changements d'état discret : si on reprend l'exemple du thermostat, la transition de ON à OFF représente l'action d'éteindre le radiateur. Les capteurs ne sont pas explicitement présents dans le modèle : la communication entre le milieu continu et le partie discrète repose sur une mémoire partagée. En effet, le modèle suppose qu'à tout instant, un état discret connaît l'ensemble du milieu extérieur, c'est-à-dire la valeur de chaque variable continue. On peut donc supposer que la partie continue et la partie discrète possède une zone de mémoire partagée sur laquelle l'environnement continu écrit et que les états discrets lisent. Cette zone mémoire représente un capteur parfait (c'est-à-dire n'introduisant pas de bruit) fonctionnant en continu. Le paradigme de communication sous sous-jacent est une communication synchrone événementielle : lorsque l'automate entre dans un nouvel état discret, il se met en attente d'un évènement lui permettant d'en sortir (soit que le prédicat de garde d'une transition est activée, soit que l'invariant de l'état est contredit). De manière très schématique, l'exécution d'un automate hybride fonctionne donc ainsi :

début
  Initialise;
  tant que *vrai* **faire**
    Attendre l'évènement E;
    si E *est une transition* **alors**
      Exécute la transition;
    sinon
      echoue ; `/* l'invariant de l'état est invalidé, aucune transition n'est activée */`
    finsi
  fintantque
fin

L'exécution de l'automate est donc bloquée dans un état discret jusqu'à ce qu'une des transitions partant de cet état soit activée. C'est donc l'évolution de l'environnement continu qui décide des instants auxquels sont exécutées les transitions discrètes. Une des conséquences majeures de ce choix est le non-déterminisme du modèle : si deux transitions sont activées au même moment, il n'est pas précisé laquelle doit être exécutée.

Le fonctionnement d'un programme critique embarqué est très différent. Ces programmes

sont dans la très grande majorité des cas générés automatiquement à partir d'un langage de spécification de haut niveau qui repose sur le modèle synchrone (par exemple SCADE). La communication entre le programme et son environnement extérieur se fait alors via des variables *volatiles* de manière périodique. Le schéma d'exécution d'un programme critique embarqué est de la forme :

début
  Initialise;
  tant que *vrai* **faire**
    Lire entrées;
    Calculer état suivant;
    Écrire sorties;
  fintantque
fin

Dans la boucle de rétroaction (Lire-Calculer-Écrire), la lecture correspond à la communication entre le milieu continu et le programme discret, alors que l'écriture correspond à l'action des actionneurs. Cette boucle est cadencée avec précision, de sorte que les données ne sont lues que périodiquement et non pas continûment. Éventuellement, une analyse de pire temps d'exécution (WCET, [FHL+01]) peut permettre d'estimer précisément la durée d'une itération de la boucle. La communication entre le milieu continu et le programme s'apparente donc plus à une communication par passage de messages qu'à une communication à mémoire partagée : à certains instants *déterminés par le programme discret*, le programme et l'environnement continu vont se synchroniser pour échanger des données. Remarquons que la communication entre le milieu continu et le programme discret s'effectue via un capteur qui introduit une discrétisation des valeurs : la valeur des variables continues réelles est discrétisée en un nombre à virgule flottante qui sera traité par le programme. Notre modélisation des capteurs prendra en compte cette double discrétisation temporelle (les capteurs ne fournissent une valeur qu'à certains instants prédéterminés) et spatiale (les valeurs fournies sont des approximations flottantes des valeurs réelles). Enfin, par rapport aux automates hybrides, l'exécution d'un programme embarqué est déterministe (car écrit dans un langage déterministe) et l'exécution du système est contrôlée d'avantage par la partie discrète que par l'environnement continu.

On voit donc que les programmes embarqués, s'il peuvent être encodés dans le formalisme des automates hybrides, reposent en fait sur un échantillonage précis du temps et des valeurs d'entrée alors que les modèles classiques de systèmes hybrides considèrent l'évolution temporelle de manière plus événementielle. Cette deuxième vision, si elle permet une modélisation plus rapide des systèmes hybrides, rend également leur validation plus compliquée : les changements d'états (c'est-à-dire les changements de mode continu) sont déterminés par des critères purement spatiaux (la valeur des variables continues), ce qui rend la détection des instants de changement d'états difficile : il faut calculer l'intersection du résultat de l'opérateur d'avancement dans le temps $\nearrow$ avec les prédicats de garde (voir section 4.4.2). Les outils d'analyse limitent donc la dynamique continue pour pouvoir détecter ces changements d'états. Dans les programmes embarqués, les changements d'états se font à travers l'écriture dans une variable volatile, à la fin du cycle Lire-Calculer-Écrire, et donc à des périodes connues. La détection de changements d'états est donc bien plus simple, et l'on pourra s'autoriser des dynamiques non-linéaires lors de l'analyse (voir chapitre 7). Deux autres considérations font que le modèle des automates hybrides est moins adapté à la modélisation et l'analyse des programmes critiques embarqués. D'une part, pour des raisons de sûreté, il sera toujours nécessaire d'analyser le code source et non pas le modèle qui l'a généré. On pourrait naturellement reconstruire, à partir du code et d'une description de l'environnement continu, un automate équivalent, mais cette solution devient rapidement impossible pour des codes de grandes tailles évoluant dans des milieux physiques complexes (typiquement, les programmes utilisés dans l'industrie aéronautique font plusieurs centaines de milliers de lignes de code et l'environnement extérieur est composé, au moins, d'une dizaine de variables continues). Par ailleurs, dans le formalisme des automates hybrides, les états discrets et les modes continus sont très liés (en fait, ce sont les mêmes) et il n'y a donc pas de séparation réelle entre le continu et le discret. Pour des projets industriels, la modélisation de l'environnement physique et du programme discret demandent des expertises très différentes et sont souvent effectuées par des équipes séparées. Il nous semble donc

préférable de disposer d'un modèle où la distinction entre le continu et le discret est nette, et tel que la communication entre les deux mondes se fassent via des interfaces définies à l'avance.

5.1.2 But du nouveau modèle

Nous avons donc développé une extension des langages de programmation impératifs qui prend en compte les particularités hybrides des programmes embarqués. Les caractéristiques principales de ce modèle sont :

1. la partie discrète est la plus proche possible des langages de programmation existant ;
2. les actions des capteurs et actionneurs sont explicitement identifiées ;
3. les parties discrètes et continues peuvent être modélisées séparément.

La première condition est nécessaire pour que l'on puisse réutiliser facilement les techniques d'analyse statique existantes et qui ont fait leurs preuves sur les codes embarqués. La deuxième condition tient au fait que dans les codes embarqués actuels, on sait à quels moments ont lieu l'acquisition des données extérieures et l'envoi de commandes vers les actionneurs. On veut donc pouvoir exprimer formellement ces informations. Enfin, la troisième condition est indispensable pour l'analyse de codes de taille industrielle. Elle est également utile pour pouvoir analyser le comportement d'un même programme dans deux environnements physiques différents à moindre coût, ou pour pouvoir comparer deux programmes censés contrôler la même grandeur physique : il suffit dans chacun de ces cas de changer une des parties du système.

Remarque (Capteurs et actionneurs) Nous veillerons, en plus des trois conditions ci-dessus, à ce que les capteurs introduisent un échantillonage spatial, c'est-à-dire une discrétisation des valeurs réelles en des valeurs représentables sur une machine, et à ce que les actionneurs ne puissent modifier que la dynamique des variables continues et non pas leurs valeurs.

Avant de décrire en détail notre nouveau modèle, faisons un petit tour d'horizon des modèles existant et expliquons pourquoi ils ne sont pas pleinement satisfaisant. Nous avons déjà vu que les automates hybrides ainsi que tous les modèles dérivées (algèbres de processus hybrides entre autres) reposent sur un modèle d'exécution très différent de celui des programmes embarqués. De plus, la séparation entre les modes discrets et continus n'est pas claire, ce qui rend compliquée et coûteuse l'étude d'un même sytème dans plusieurs environnements continus différents. Nous ne pouvons donc pas utiliser les automates hybrides pour l'analyse des programmes embarqués. Le logiciel le plus utilisée pour décrire et simuler un système hybride est Simulink. En Simulink, il n'y a pas de différence entre un système discret et un système continu, si bien qu'on peut décrire toute sorte de systèmes. Cependant, il n'existe pas de sémantique formelle de Simulink de sorte que le comportement exact du système n'est pas connu, et le mélange entre les comportements discrets et continus fait que ce modèle ne peut être utilisé pour la vérification de programmes industriels.

Le langage Modelica [OEM99] est un langage de haut niveau orienté objet permettant de décrire de gros systèmes industriels complexes. C'est un équivalent textuel de Simulink, et on peut grâce à Modelica modéliser des systèmes ayant des comportements discrets et continus. Les sytèmes sont décrits comme des ensembles d'équations, et le simulateur associé a pour but de résoudres ces équations. De nombreuses bibliothèques permettent d'inclure dans le système des éléments méchaniques, électriques, thermiques, hydrauliques... Cette trop grande expressivité est cependant un défault quand on s'intéresse à la vérification des programmes embarqués : le langage mélange les systèmes discrets et continus sans véritable séparation et comme pour Simulink, la sémantique globale du langage n'est pas clairement définie.

Le modèle qui nous semble le plus proche d'un modèle utilisable pour l'analyse de gros systèmes industriels est le langage synERGY [BPS06]. synERGY est un langage orienté objet intégrant un paradigme de programmation synchrone proche de Lustre. Il est pensé pour décrire efficacement les programmes embarqués et sa sémantique est précisément décrite. Un compilateur de synERGIE vers le langage C a été développé dans le but de produire un code embarqué le plus efficace possible tant du point de vue de l'espace mémoire occupé que de la rapidité d'exécution. Cependant,

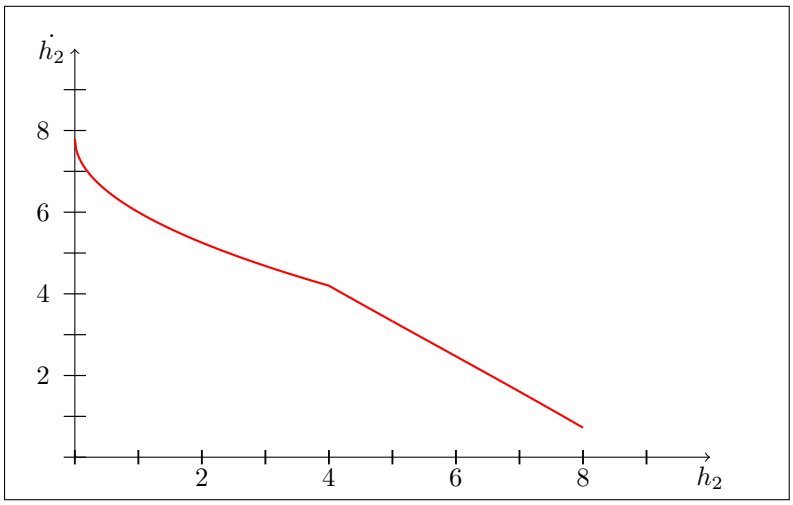


FIG. 5.1 – Fonction donnant $\dot{h_2}$ en fonction de h_2 dans le cas où les deux vannes sont ouvertes (équation (1.2)), avec $H = 4$, $k_1 = 2.6$, $k_2 = 1.8$ et $h_1 = 9$.

synERGIE n'est pas à proprement parler un langage hybride car il ne permet pas de décrire l'environnement continu avec lequel le programme interagit via des équations différentielles par exemple. Il doit donc être étendu pour prendre en compte cet aspect, et nous avons choisi de fonder notre nouveau modèle sur un langage de bas niveau (par exemple, le code C produit par synERGIE) plutôt que de rester au niveau de la spécification. Ce choix nous permet de rester proche des codes existants et donc d'appliquer rapidement nos techniques à des problèmes industriels.

5.2 Description du modèle

5.2.1 Modélisation de la partie continue

La partie continue des systèmes hybrides que nous considérons est formée de variables évoluant continûment dans le temps et qui représentent les grandeurs physiques dont le programme a besoin (ou sur lesquelles il agit). La dynamique de ces variables peut être modifiée de deux façons : soit par un changement dû au programme via les actionneurs, soit par un changement dû à la dynamique elle-même, par exemple lorsque l'eau passe au-dessus du premier tuyau horizontal dans le système des deux réservoirs (voir section 1.6). Notre modèle doit être assez expressif pour intégrer ces deux phénomènes. Pour cela, nous prenons comme point de départ les problèmes de Cauchy comme définis en section 3.1. L'expressivité du modèle continu dépend alors du type de fonctions F que nous autorisons dans l'équation (3.1). Remarquons que le choix de fonder la modélisation du milieu extérieur des programmes embarqués sur des EDOs n'est pas anodin. On peut en effet effectuer un parallèle fort entre une équation différentielle et le code source d'un programme informatique. Les deux sont des représentations d'un système dynamique : le code source est un modèle pour le programme exécutable obtenu après compilation, et chaque instruction relie l'état du programme à un instant t (c'est-à-dire avant l'exécution de l'instruction) à l'état du programme à l'instant $t+1$ (après son exécution). Une équation différentielle est elle un modèle pour une fonction continue : la fonction F décrit l'état du système à l'instant $t + dt$ en fonction de l'état du système à l'instant t. Une EDO joue le même rôle qu'un programme mais pour des systèmes dynamiques continus, il est donc naturel de les utiliser comme base pour modéliser le milieu continu des programmes embarqués.

Pour représenter les changements dus à l'environnement lui-même, nous supposerons que la fonction F est continue et Lipschitz par morceaux. Cela permet de définir des dynamiques différentes en différentes régions de l'espace. Rappelons en effet qu'une fonction $g : \mathbb{R} \to \mathbb{R}^n$ est

Lipschitz par morceaux si et seulement s'il existe $x_0 < x_1 < \cdots < x_n \in \mathbb{R} \cup \{-\infty, \infty\}$ tels que la restriction de g à $[x_i, x_{i+1}]$ est Lipschitz pour tout i (cette définition s'étend naturellement aux fonctions définies sur $\mathbb{R}^m$). Par exemple, la fonction donnant la dynamique de la hauteur d'eau dans le deuxième réservoir (voir la figure 1.6) est une fonction Lipschitz par morceaux (voir la figure 5.1). Dans ce cas, la fonction F est continue et globalement Lipschitz, de sorte que la solution du problème de Cauchy sous-jacent existe et est unique. En particulier, le théorème de convergence des itérées de Picard (théorème 3.2) est valable. L'hypothèse de n'avoir que des fonctions Lipschitz est forte, mais elle a du sens si on se rappelle que les équations différentielles que l'on va considérer sont des lois de la physique. Tous les phénomènes décrits seront donc très réguliers et l'hypothèse (très classique lorsque l'on utilise des équations différentielles) d'une condition de Lipschitz globale nous apparaît comme raisonnable.

Pour prendre en compte les changements occasionnés par le programme discret, nous introduisons dans la fonction F un vecteur de paramètres formels k représentant l'effet de m actionneurs sur le milieu continu. On aura donc $F = F(y, k)$; chaque coordonnée de k représente l'état d'un des actionneurs, et pourra être modifiée par le programme : k prendra ses valeurs dans $\mathbb{B}^m$ (c'est-à-dire l'ensemble des vecteurs de valeurs booléennes de dimension m), et si la i-ième coordonnée de k vaut 1, cela voudra dire que le i-ième actionneur est activé. On dispose ainsi de 2^m modes continus possibles, et pour toute valeur possible des paramètres $k \in \mathbb{B}^m$, nous noterons F_k la fonction telle que $\forall y \in \mathbb{R}^n,\ F_k(y) = F(y, k)$.

Définition 5.1 (Modèle du milieu continu) Un modèle κ pour le milieu continu est un quadruplé $\kappa = \big(Var, m, \{F_k\}_{k\in\mathbb{B}^m}, (y_0, k_0)\big)$ tel que :

1. Var est un ensemble de noms pour les variables continues définies par les équations différentielles ;
2. $m \in \mathbb{N}$ est le nombre d'actionneurs disponibles ;
3. $y_0 \in \mathbb{R}^n$ est une condition initiale, avec $n = |Var|$;
4. $k_0 \in \mathbb{B}^m$ est le mode continu initial ;
5. $\forall k \in \mathbb{B}^m$, $F_k : \mathbb{R}^n \to \mathbb{R}^n$ est une fonction continue Lipschitz par morceaux, avec $n = |Var|$.

Exemple 5.1 (Problème du thermostat) Le système du thermostat (voir section 4.1) possède un actionneur, et donc deux modes continus. La partie continue du système est donnée par $\kappa_t = \big(\{x\}, 1, \{F_0, F_1\}, (x_0, k_0)\big)$ avec :

$$F_0(x) = -x/3 \qquad F_1(x) = 9 - x/3\ . \tag{5.1}$$

Les équations différentielles donnant l'évolution continue seront donc $\dot{x} = F_0(x)$ et $\dot{x} = F_1(x)$ selon le mode choisi.

Exemple 5.2 (Problème des deux réservoirs) Le problème des deux réservoirs possède quatre modes continus correspondant aux cas où les vannes 1 et 2 sont ouvertes et/ou fermées. Un modèle continu pour ce système sera donc $\kappa_r = \big(\{h_1, h_2\}, 2, \{F_{0,0}, F_{1,0}, F_{0,1}, F_{1,1}\}, (y_0, k_0)\big)$ où $F_{0,0}$ représente le cas où les deux vannes sont ouvertes (équation (1.2)), $F_{1,0}$ le cas où la vanne 1 est fermée (équation (1.3)), $F_{0,1}$ le cas où la vanne 2 est fermée (équation (1.4)) et $F_{1,1}$ le cas où les deux vannes sont fermées (équation (1.5)).

5.2.2 Modélisation de la partie discrète

La partie discrète de notre modèle est un programme écrit dans une extension du langage **SIMPLE** décrit à la section 2.2.1. En plus des expressions arithmétiques et des instructions discrètes classiques, nous introduisons un nouveau type de variables, les variables continues ($CVar$), et trois instructions hybrides (**HStmt** dans la figure 5.2). Ces trois instructions sont :

- ***sens***.$y?X$ où $y \in CVar$ est une variable continue et $X \in Var$ est une variable discrète. Cette instruction sert à modéliser l'effet d'un capteur qui alloue à la variable discrète X la valeur de la variable continue y à l'instant où l'instruction est exécutée.
- ***act***.$k!c$ où $c \in \{0, 1\}$ et $k \in \mathbb{N}$. Cette instruction représente l'effet du k-ième actionneur sur l'environnement physique : la dynamique du milieu continu va être modifiée et sera régie, après l'exécution de l'instruction, par la fonction F_l, où $l \in \mathbb{B}^m$ est telle que la k-ième coordonnée de l vaut c.
- ***wait*** u où $u \in \mathbb{R}_+$. Cette instruction sera utilisée pour symboliser la durée des autres instructions discrètes : on suppose en effet que chaque instruction est instantanée et on ajoute explicitement des instructions ***wait*** pour compenser le fait qu'elles ne l'étaient pas.

Remarque Dans l'instruction de délai ***wait***, nous supposons que le délai est un valeur constante $u \in \mathbb{R}_+$. Nous n'autorisons pas à ce que le délai soit une variable ou une expression arithmétique pour empêcher les effets Zénon. En effet, en autorisant le délai à être une expression arithmétique, on pourrait écrire le programme

$$\textbf{\textit{while}}\ 1\ \textbf{\textit{do}}\ \big(\textbf{\textit{wait}}\ \big(1/(n*n)\big);\ \textbf{\textit{act}}.1!0;\ n := n+1;\big)$$

qui effectue une infinité d'actions en un temps fini. Étant donné que l'instruction ***wait*** attend un constante $u \in \mathbb{R}_+$ comme argument, et comme les programmes sont de tailles finies, il existe un temps minimum $\tau > 0$ telle que toutes les instructions ***wait*** augmentent le temps d'au moins τ secondes. La seule possibilité d'avoir une exécution infinie qui ne dure qu'un temps fini est donc d'avoir une exécution infinie n'exécutant qu'un nombre fini d'instructions ***wait***.

Donc, la seule possibilité d'obtenir un effet Zénon est d'avoir une boucle infinie telle qu'aucune des itérations de la boucle ne contienne d'instructions ***wait***. Le programme effectuerait alors une infinité d'actions en un temps nul. Une condition nécessaire (mais pas suffisante) pour ne pas avoir d'effets Zénon est donc qu'il y ait une instruction ***wait*** dans le corps de chaque boucle infinie. Cette condition n'est cependant pas suffisante (l'instruction peut ne pas être exécutée), et une analyse de vivacité permettrait de prouver l'absence d'effet Zénon : il faut montrer que l'instruction ***wait*** sera exécutée une infinité de fois. Des outils comme Terminator [CPR06b, CGP+07] permettent de le faire.

Nous nommerons cette extension de **SIMPLE** pour le calcul hybride **H-SIMPLE**. Cette extension est très proche du langage initial (sa syntaxe complète est donnée à la figure 5.2), de sorte qu'il est facile de traduire un programme écrit en **SIMPLE** en un programme de **H-SIMPLE** (en particulier, tout programme de **SIMPLE** est un programme de **H-SIMPLE**). Les rajouts sont liés aux caractéristiques hybrides de ces programmes (communication avec le milieu extérieur et notion de temps d'exécution), et sont en fait souvent présent, sous forme de commentaires, dans les codes critiques générés depuis un langage de spécification. Il n'est en effet pas rare de voir, au début d'une boucle, un commentaire indiquant la fréquence à laquelle cette boucle s'exécute. Dans ces cas, il est très simple d'ajouter une instruction ***wait*** à la fin de la boucle pour simuler ce temps d'exécution.

Remarquons que les choix que nous avons effectués pour encoder les échanges entre le programme et l'environnement continu ne sont pas anodins. Nous voyons en effet le programme et l'environnement comme deux processus dynamiques s'exécutant en parallèle, les capteurs et les actionneurs étant les canaux de communication entre les deux. Parmi l'ensemble des modèles de processus communiquant existant, celui qui se rapproche le plus du type de comminucation considéré ici est le langage CSP (*Communicating Sequential Processes*) de Hoare [Hoa85]. En effet, les canaux de communication sont connus à l'avance et fixes (le système ne crée pas de nouveaux actionneurs ou capteurs dynamiquement). Nous avons donc choisi des instructions qui utilisent la même syntaxe que les instructions CSP : le "?" modélise la réception de valeur, le "!" l'envoi de message.

Définition 5.2 (Modèle de la partie discrète) Un modèle Δ pour la partie discrète est un triplé $(CVar, m, P)$ tel que :

$$f \in \mathbb{F} \qquad X \in Var$$

$$\begin{array}{rrcl} \textbf{AExp :} & a & ::= & f \mid X \mid a + a \mid a - a \mid a \times a \mid a/a \\ \textbf{BExp :} & b & ::= & \textbf{\textit{true}} \mid \textbf{\textit{false}} \mid a = a \mid a \leq a \\ \textbf{Stmt :} & s & ::= & X := a \mid s; s \mid \textbf{\textit{if}}\ b\ \textbf{\textit{then}}\ s\ \textbf{\textit{else}}\ s \mid \textbf{\textit{while}}\ b\ \textbf{\textit{do}}\ s \mid hs \end{array}$$

$$u \in \mathbb{R}_+ \quad X \in Var \quad y \in CVar \quad k \in \mathbb{N} \quad c \in \{0, 1\}$$

$$\textbf{HStmt :} \quad hs \quad ::= \quad \textbf{\textit{sens}}.y?X \mid \textbf{\textit{act}}.k!c \mid \textbf{\textit{wait}}\ u$$

FIG. 5.2 – Syntaxe du langage **H-SIMPLE**.

1. $CVar$ est un ensemble de variables continues ;
2. $m \in \mathbb{N}$ est le nombre d'actionneurs disponibles ;
3. P est un programme écrit dans le langage **H-SIMPLE** n'utilisant que des variables continues de $CVar$ pour les actions ***sens*** et que des entiers $k \in [1, m]$ pour les actions ***act***.

Exemple 5.3 (Problème des deux réservoirs.) La partie discrète du problème des deux réservoirs (c'est-à-dire le contrôleur) s'encode très simplement en **H-SIMPLE** (figure 5.3). Le contrôleur mesure les hauteurs d'eau h_1 et h_2 (lignes 4 et 5), puis, si les niveaux critiques ont été franchis, il actionne les vannes et déclenche une alarme indiquant un problème critique dans le système. On a rajouté dans le programme de la figure 5.3 deux fonctions d'anticipation (`anticipate_h1` et `anticipate_h2`, lignes 14 et 15) qui permettent de prédire la hauteur d'eau dans chacun des deux réservoirs deux secondes dans le futur. On ne précise pas l'implémentation de cette fonction d'anticipation, on peut utiliser par exemple une extrapolation linéaire. En utilisant les valeurs calculées, on peut alors contrôler plus finement les niveaux d'eau en anticipant sur le temps de fermeture des vannes. On va donc ouvrir (ou fermer) les vannes si la valeur prédite dépasse les seuils critiques (lignes 16 à 23). En fin de boucle, l'instruction ***wait*** permet de simuler la durée d'exécution des instructions précédentes (ici, 0.01 secondes).

5.2.3 Modélisation du système hybride

Dans ce formalisme, un système hybride sera donc un couple (Δ, κ) tel que Δ est un modèle de la partie discrète et κ un modèle de la partie continue. Nous imposerons néanmoins que les deux modèles partagent les mêmes variables continues et les mêmes actionneurs.

Définition 5.3 (Modèle du système hybride) Un modèle Ω pour le système hybride est un couple $\Omega = (\Delta, \kappa)$ tel que :
- $\Delta = \big(CVar, m, P\big)$ est un modèle de la partie discrète ;
- $\kappa = \big(Var, m', \{F_k\}_{k \in \mathbb{B}^{m'}}, (y_0, k_0)\big)$ est un modèle de la partie continue ;
- $CVar \subseteq Var$ et $m \leq m'$.

La dernière condition permet de s'assurer que le programme et le milieu continu sont compatibles, c'est-à-dire que les variables sur lesquelles le programme peut agir (via une instruction ***sens*** ou ***act***) sont bien des variables contrôlées par le milieu continu.

Remarquons que ce modèle de systèmes hybrides est conforme aux objectifs que nous nous étions fixés :

```
1  int main() {
2    sensor x1,x2;          // capteurs
3    while (true) {
4      sens.x1?h1;
5      sens.x2?h2;
6      if (h1>h1_max)
7        act.1!0; throw( alarm );
8      if (h2>h2_max)
9        act.2!0; throw( alarm );
10     if (h1<h1_min)
11       act.1!1; throw( alarm );
12     if (h2<h2_min)
13       act.2!1; throw( alarm );
14     h1_in_2_secs = anticipate_h1(h1,h2);
15     h2_in_2_secs = anticipate_h2(h1,h2);
16     if (h1_in_2_secs>h1_max)
17       act.1!0;
18     if (h2_in_2_secs>h2_max)
19       act.2!0;
20     if (h1_in_2_secs<h1_min)
21       act.1!1;
22     if (h2_in_2_secs<h2_min)
23       act.2!1;
24     wait (0.01);   // delai
25   }
26 }
```

FIG. 5.3 – Encodage du système des deux réservoirs en **H-SIMPLE**.

1. une modélisation de la partie discrète ne demande que quelques ajouts aux programmes existants ;
2. les actions des capteurs et des actionneurs sont explicites, et la communication entre les parties discrètes et continues se fait par envoi de messages ;
3. les sous-systèmes discrets et continus peuvent être modélisés séparément, dans des formalismes différents.

La figure 1.4 peut maintenant être étendue pour rendre compte de notre nouveau modèle. La figure 5.4 décrit plus précisément le fonctionnement des programmes embarqués.

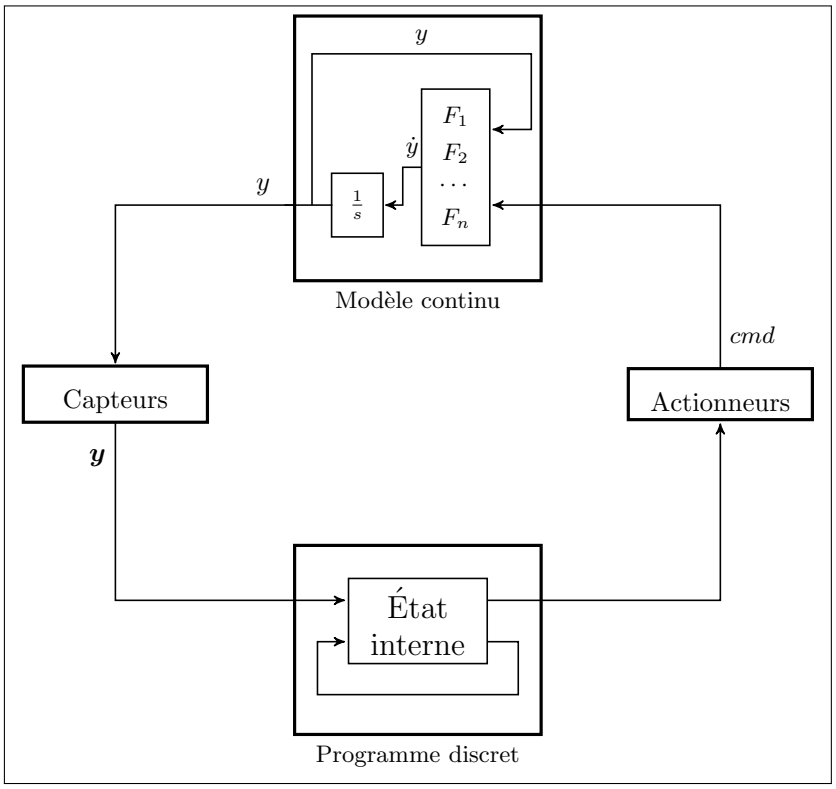


FIG. 5.4 – Fonctionnement d'un programme embarqué. La partie discrète est un programme écrit en **H-SIMPLE** qui calcule, en fonction de son état interne et des entrées provenant des capteurs, un résultat envoyé aux actionneurs. Ce résultat, noté *cmd* sur le schéma, équivaut à choisir, parmi tous les modes disponibles, celui correspondant à l'actionneur à activer. Le milieu continu peut être vu comme une EDO à commutateur représentée par le bloc en haut du schéma. L'entrée de ce bloc est une commande venant des actionneurs, et la sortie est la valeur courante y de la solution de l'EDO. Cette valeur est transmise au capteur qui la transforme en le nombre flottant le plus proche, c'est-à-dire $\boldsymbol{y}$.

5.2.4 Construction de la sémantique du modèle

Nous cherchons à définir une sémantique dénotationnelle des systèmes hybrides décrits dans notre modèle. La difficulté dans cette tâche tient à l'hétérogénéité du système et à la séparation entre les parties discrètes et continues. Le comportement de chacun des sous-systèmes se calcule en effet de manière très différente. Pour le programme, on utilise généralement une sémantique dénotationnelle comme présenté en section 2.2. Cette sémantique définit le comportement d'un programme comme une fonction calculée à l'aide d'un théorème de point fixe à la Kleene, c'est-à-dire un théorème portant sur des fonctions monotones dans un CPO. Pour l'équation différentielle,

sa sémantique est sa solution (voir section 3.1), et on montre son existence via un théorème de point fixe à la Banach (théorème 3.2), c'est-à-dire un théorème portant sur des fonctions contractantes dans un espace métrique. Le but de la sémantique que nous proposons est d'unifier ces deux descriptions. Pour cela, nous choisissons de définir la sémantique d'une équation différentielle de la même manière que l'on définit la sémantique d'un programme, c'est-à-dire à travers un théorème de point-fixe portant sur une notion d'ordre et non pas de distance.

Notre construction de la sémantique se fera en trois étapes. Dans un premier temps, nous supposerons que le comportement de la partie discrète est complètement connu, et nous définirons la sémantique de la partie continue (section 5.3). Cela revient à définir la sémantique d'un problème de Cauchy, et nous devrons donc définir un CPO et une fonction sur ce CPO telle que la solution du problème de Cauchy soit le point fixe de cette fonction. Dans un deuxième temps, nous définirons la sémantique des systèmes hybrides en supposant que l'on dispose de fonctions à la place des équations différentielles (section 5.4). Cela revient à dire que l'on suppose connue l'évolution du milieu continu et on calcule alors une sémantique pour **H-SIMPLE**. Enfin, nous unifierons les deux résultats obtenus pour définir la sémantique du système hybride comme une combinaison des sémantiques continues et discrètes (section 5.5).

5.3 Sémantique dénotationnelle de la partie continue

Pour simplifier la lecture de cette section, nous nous limitons au cas où la partie continue d'un système hybride est composée d'une seule variable continue. En particulier, les fonctions F_k de la définition 5.1 sont des fonctions d'une seule variable réelle. Cependant, tous les résultats que nous présentons s'étendent sans aucun problème au cas de n variables continues ; en particulier, pour toutes les fonctions définies dans cette section (largeur, composition, primitive, voir section 5.3.2), leurs extensions composante par composante aux dimensions supérieures possèdent les mêmes propriétés que dans le cas de la dimension 1.

Nous commençons donc par donner une sémantique dénotationnelle à la partie continue des systèmes hybrides en supposant connue la sémantique de la partie discrète. L'influence du programme sur l'environnement continu se fait via des changements de dynamique au moyen des actionneurs ; si on suppose connu le comportement du programme, cela revient à dire que l'on suppose connue la suite des actions effectuées par le programme ainsi que les instants auxquels ces actions ont eu lieu. On voit donc le programme comme une suite d'actions, autrement dit on voit le programme comme une fonction $s : \mathbb{R}_+ \to \mathbb{B}^m$ constante par morceaux dont les valeurs sont les commandes envoyées aux actionneurs. Remarquons que, puisque l'instruction ***wait*** attend un argument u constant, il existe un temps minimum $\tau > 0$ entre deux discontinuités dans la fonction s. On ne peut donc pas avoir une fonction constante par morceaux avec des marches de plus en plus petites. La sémantique d'un modèle continu $\kappa = \big(CVar, m, \{F_k\}_{k\in\mathbb{B}^m}, (y_0, k_0)\big)$ sous l'effet de la fonction s, que nous noterons $[\![\kappa]\!]_s$, est alors la solution de l'équation différentielle $\dot{y} = F(y,t)$, où F est la fonction définie par :

$$\forall t \in \mathbb{R}_+,\ \forall y \in \mathbb{R},\ F(y,t) = F_{s(t)}(y)\ . \tag{5.2}$$

Autrement dit, F est une fonction continue par morceaux qui coïncide avec la fonction F_k aux instants t où $s(t) = k$. Nous noterons $[\![F, y_0]\!]^c$ la solution du problème de Cauchy $\dot{y} = F(y),\ y(0) = y_0$.

Définition 5.4 (Sémantique continue) Soit κ un modèle de la partie continue, et $s : \mathbb{R}_+ \to \mathbb{B}^m$ une fonction constante par morceaux représentant le programme discret. La sémantique de κ par rapport à s est la solution du problème de Cauchy $\dot{y} = F(y,t),\ y(0) = y_0$ où F est définie par l'équation (5.2). On notera :

$$[\![\kappa]\!]_s = [\![F, y_0]\!]^c \text{ avec } \forall y \in \mathbb{R},\ \forall t \in \mathbb{R}_+,\ F(y,t) = F_{s(t)}(y)\ . \tag{5.3}$$

Remarque Dans l'équation différentielle de la définition 5.4, la fonction F n'est que continue par morceaux, mais elle est globalement Lipschitz en y. En effet, chaque fonction F_k est globalement Lipschitz, et donc la plus grande des constantes de Lipschitz des fonctions F_k est une constante de Lipschitz pour F. De plus, pour tout $t \in \mathbb{R}_+$, la fonction $\lambda y.F(y,t)$ est continue et il existe un temps minimum $\tau > 0$ entre deux discontinuités de F. Dans ce cas, la solution de l'équation différentielle existe et est unique et le théorème de Picard (théorème 3.2) est vrai. Nous noterons $\mathcal{C}^0_m(\mathbb{R})$ l'ensemble des fonctions $F(y,t)$ définies sur $\mathbb{R} \times \mathbb{R}_+$ à valeurs dans $\mathbb{R}$ et telles que F est continue par morceaux, globalement Lipschitz et continue en y, et telle qu'il existe un temps minimum non nul entre deux points de discontinuité. Remarquons que cela revient à dire que l'ensemble des points de discontinuité est dénombrable.

Dans la suite, nous nous attachons à définir $[\![F, y_0]\!]^c$, c'est-à-dire que nous construisons la solution du problème de Cauchy $\dot{y} = F(y,t),\ y(0) = y_0$. D'après le théorème 3.2, cette solution est obtenue comme le point fixe, au sens de la théorie du point fixe de Banach, de l'opérateur de Picard (équation (3.3)). Nous allons traduire ce théorème de point fixe en un théorème de point fixe à la Kleene, c'est-à-dire un point fixe portant sur une notion d'ordre et non plus de distance. Nous devons donc définir une structure d'ordre et un opérateur sur cette structure dont $[\![F, y_0]\!]^c$ soit le point fixe. Pour cela, nous définissons un domaine de fonctions à *information partielle*, c'est-à-dire des fonctions dont la valeur en chaque point $x \in \mathbb{R}_+$ est un intervalle. Nous dirons alors qu'une fonction f porte plus d'information qu'une fonction g si elle est incluse dans g en tout point. Ainsi, les éléments maximaux seront les fonctions à valeur réelle, c'est-à-dire les fonctions associant à tout point x un singleton. La sémantique continue $[\![F, y_0]\!]^c$ va construire un élément maximal (la solution y_∞ de l'équation différentielle de la définition 5.4) comme la limite d'une suite d'approximations, c'est-à-dire comme la limite d'une suite de fonctions à valeur intervalle. Les sections suivantes formalisent cette intuition.

5.3.1 Treillis des fonctions continues intervalles

Rappelons que $I_\mathbb{R}$ représente l'ensemble des intervalles réels fermés.

Définition 5.5 (Fonctions intervalles) Nous noterons $\mathcal{IF}_\infty$ l'ensemble des fonctions intervalles définies sur $\mathbb{R}_+$:

$$\mathcal{IF}_\infty = \{f : \mathbb{R}_+ \to I_\mathbb{R}\}\ .$$

Une fonction $f \in \mathcal{IF}_\infty$ associe donc à chaque instant $x \in \mathbb{R}_+$un intervalle $f(t) \in I_\mathbb{R}$. Nous représenterons graphiquement une telle fonction comme montré à la figure 5.5. Cette figure montre deux fonctions de $\mathcal{IF}_\infty$: la fonction rouge est la fonction constante f telle que $\forall x \in \mathbb{R}_+,\ f(x) = [0,1]$ et la fonction bleue est la fonction g telle que $\forall x \in \mathbb{R}_+,\ g(x) = \left[-\frac{1}{x+1}, 1 + \frac{1}{x+1}\right]$.

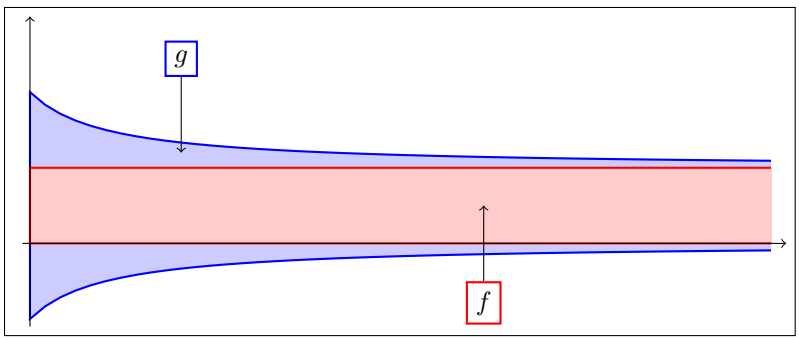


FIG. 5.5 – Représentation graphique des fonctions de $\mathcal{IF}_\infty$.

On munit $\mathcal{IF}_\infty$ d'un ordre partiel qui est une extension point à point de l'ordre inverse sur les intervalles. Rappelons que l'ordre inverse $\sqsupseteq$ sur les intervalles est donné par $\forall x, y \in I_\mathbb{R},\ x \sqsupseteq y \Leftrightarrow y \subseteq x$. Le domaine $\left(I_\mathbb{R}, \sqsupseteq\right)$ est un CPO.

Définition 5.6 (Ordre partiel sur $\mathcal{IF}_\infty$) Soient $f, g \in \mathcal{IF}_\infty$ deux fonctions intervalles. On a alors :

$$f \sqsubseteq_\infty g \Leftrightarrow \forall x \in \mathbb{R}_+,\ f(x) \sqsupseteq g(x) \Leftrightarrow \forall x \in \mathbb{R}_+,\ g(x) \subseteq f(x)\ . \tag{5.4}$$

L'ordre $\sqsubseteq_\infty$ signifie qu'en tout point $x \in \mathbb{R}_+$, la fonction g apporte plus d'information que f. La notion d'information que nous utilisons ici est contre-intuitive (on dira qu'un intervalle contient plus d'information s'il est plus petit), mais est cohérente avec l'idée que ces fonctions intervalles sont des approximations de la solution de l'équation différentielle. Cette solution est en effet une fonction réelle, c'est-à-dire une fonction intervalle de largeur nulle, et elle est censée représenter l'élément d'information maximale. Il faut donc que cette fonction soit plus grande que toutes ses approximations, ce qui impose d'utiliser l'ordre inverse sur les intervalles.

Exemple 5.4 Sur la figure 5.5, la fonction rouge (f) est plus grande que la fonction bleue (g).

Définition 5.7 (Fonctions supérieures et inférieures) Soit $f \in \mathcal{IF}_\infty$. On définit les fonctions supérieures $\overline{f} : \mathbb{R}_+ \to \mathbb{R}$ et inférieure $\underline{f} : \mathbb{R}_+ \to \mathbb{R}$ comme les deux fonctions réelles telles que :

$$\forall x \in \mathbb{R}_+,\quad f(x) = \left[\underline{f}(x), \overline{f}(x)\right]\ .$$

Exemple 5.5 Pour la fonction g de la figure 5.5, on a :

$$\forall x \in \mathbb{R}_+,\quad \underline{g}(x) = -\frac{1}{x+1} \quad \overline{g}(x) = 1 + \frac{1}{x+1}\ .$$

Nous allons naturellement nous limiter à l'étude des fonctions de $\mathcal{IF}_\infty$ qui possèdent un certain degré de continuité. Nous allons en effet définir la notion de primitive pour les fonctions intervalles, ce qui suppose une certaine régularité des fonctions inférieures et supérieures. Nous travaillerons donc uniquement sur les fonctions intervalles dont la fonction supérieure (respectivement inférieure) est semi-continue supérieurement (respectivement inférieurement). Rappelons qu'une fonction $f : \mathbb{R}_+ \to \mathbb{R}$ est dite semi-continue supérieurement si et seulement si :

$$\forall x_0 \in \mathbb{R}_+,\ \forall \epsilon > 0,\ \text{il existe un voisinage } U \text{ de } x_0 \text{ tel que } \forall x \in U,\ f(x) \leq f(x_0) + \epsilon\ . \tag{5.5}$$

La semi-continuité inférieure est équivalente sauf que l'on impose que $\forall x \in U,\ f(x) \geq f(x_0) - \epsilon$. Pour une fonction $f \in \mathcal{IF}_\infty$, lorsque les fonctions $\underline{f}$ et $\overline{f}$ sont semi-continues supérieurement et inférieurement, respectivement, on dira que f est continue. En fait, dans ce cas, f peut-être vue comme une fonction continue au sens de la topologie de Scott du domaine $(\mathbb{R}_+, \leq)$ ($\leq$ est ici l'ordre naturel sur les nombres réels) vers $(I_\mathbb{R}, \sqsupseteq)$ [EL02]. Nous présentons ici un cadre légèrement différent et plus simple que celui décrit par Edalat et Lieutier dans [EL02] ou Edalat et Pattinson dans [EP04], mais nous utiliserons certains de leurs résultats, et donnerons des preuves légèrement différentes de celles apportées par Edalat.

Définition 5.8 (Fonctions intervalles continues) Soit $f \in \mathcal{IF}_\infty$. On dira que f est continue si et seulement si $\underline{f}$ est semi-continue inférieurement et $\overline{f}$ semi-continues supérieurement. Nous noterons $\mathcal{IF}^0_\infty$ l'ensemble des fonctions intervalles continues :

$$\mathcal{IF}^0_\infty = \left\{ f \in \mathcal{IF}_\infty\ :\ f \text{ est continue} \right\}\ .$$

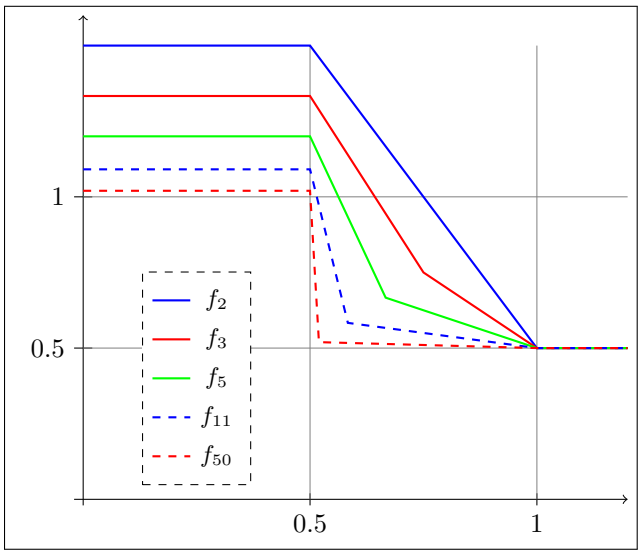


FIG. 5.6 – Représentation des fonctions f_n de l'équation 5.6.

Proposition 5.1 ([EL02]) *Le domaine $\mathcal{IF}^0_\infty$ muni de l'ordre $\sqsubseteq_\infty$ (définition 5.6) est un CPO.*

Preuve Les domaines $\big(\mathbb{R}_+, \leq\big)$ et $\big(I_\mathbb{R}, \sqsupseteq\big)$ sont des CPOs et les fonctions continues sont des fonctions Scott-continues (au sens de la topologie euclidienne) de $\big(\mathbb{R}_+, \leq\big)$ vers $\big(I_\mathbb{R}, \sqsupseteq\big)$, et l'ensemble des fonctions Scott-continues entre deux CPOs est un CPO [GHK+03]. □

Remarque (Condition de semi-continuité) Remarquons que la condition de semi-continuité inférieure et supérieure pour les fonctions intervalles est nécessaire pour que $\mathcal{IF}^0_\infty$ soit un CPO. En effet, si on impose des conditions plus fortes, par exemple, que pour tout $f \in \mathcal{IF}_\infty$, $\underline{f}$ et $\overline{f}$ sont continues, alors $\mathcal{IF}^0_\infty$ n'est plus un CPO. Considérons par exemple la suite de fonctions linéaires par morceaux $\big(\overline{f_n}\big)_{n\in\mathbb{N}}$ telles que $\forall n \in \mathbb{N}$:

$$\overline{f_n}(0) = \overline{f_n}(\frac{1}{2}) = 1 + \frac{1}{n}, \quad \overline{f_n}(\frac{1}{2} + \frac{1}{n+1}) = \frac{1}{2} + \frac{1}{n+1}, \quad \overline{f_n}(1) = \frac{1}{2}\,. \tag{5.6}$$

La figure 5.6 montre quelques unes de ces fonctions. La suite $\big(f_n\big)_{n\in\mathbb{N}}$ définie par $\forall x \in \mathbb{R}_+,\ f_n(x) = [0, \overline{f_n}(x)]$ vérifie $\forall n \in \mathbb{N},\ f_n \sqsubseteq_\infty f_{n+1}$. Posons $f = \sqcup_{n\in\mathbb{N}} f_n$. Clairement, $\overline{f}$ vérifie $\forall x \in [0, \frac{1}{2}],\ \overline{f}(x) = 1$ et $\forall x \in]\frac{1}{2}, +\infty[,\ \overline{f}(x) = \frac{1}{2}$. Donc, $\overline{f}$ n'est pas continue en $\frac{1}{2}$, par contre elle est semi-continue supérieurement.

Remarque (Fonctions réelles) Notons également que l'ensemble $\mathcal{C}^0(\mathbb{R}_+)$ des fonctions continues réelles définies sur $\mathbb{R}_+$ est inclus dans $\mathcal{IF}^0_\infty$ via la fonction $\gamma : \mathcal{C}^0(\mathbb{R}_+) \to \mathcal{IF}^0_\infty$ telle que $\forall f \in \mathcal{C}^0(\mathbb{R}_+),\ \gamma(f) = \lambda x.[f(x), f(x)]$. La fonction γ est injective, et nous dirons, par abus de notation, que $f \in \mathcal{IF}^0_\infty$ pour tout $f \in \mathcal{C}^0(\mathbb{R}_+)$.

Rappelons que nous cherchons à définir la solution de l'équation différentielle de la définition 5.4. Cette solution est une fonction réelle définie sur $\mathbb{R}_+$ que nous allons calculer comme la limite d'une série d'approximations. Chaque approximation sera une fonction intervalle continue, et nous allons améliorer cette approximation "de la gauche vers la droite" : on voit chaque instant $t \in \mathbb{R}_+$ comme un point de contrôle pour la fonction, et, comme c'est le cas pour la sémantique des programmes, nous allons mettre à jour les points de contrôle en commençant par les plus proches du point initial (dans notre cas il s'agit de 0). Nous devons donc ajouter à une fonction $f \in \mathcal{IF}^0_\infty$ une information indiquant jusqu'à quel instant cette approximation est valide, c'est-à-dire jusqu'à quel instant nous l'avons déjà calculée. Nous allons donc considérer des couples (f, t) tels que $f \in \mathcal{IF}^0_\infty$ et $t \in \overline{\mathbb{R}_+}$, où $\overline{\mathbb{R}_+} = \mathbb{R}_+ \cup \{+\infty\}$ est le treillis complet des réels positifs.

Intuitivement, un couple (f,t) représente donc une approximation d'une fonction réelle qui est significative (ou représentative) jusqu'au temps t. Nous dirons que t est le *support* de f.

Définition 5.9 (Fonctions intervalles continues avec support) Nous noterons $\mathcal{D}^0$ le produit cartésien de $\mathcal{IF}^0_\infty$ et de $\overline{\mathbb{R}_+}$, le treillis complet des nombres réels positifs muni de l'élément $+\infty$.

$$\mathcal{D}^0 = \mathcal{IF}^0_\infty \times \overline{\mathbb{R}_+} \ . \tag{5.7}$$

Définition 5.10 (Ordre partiel sur $\mathcal{D}^0$) On étend l'ordre $\sqsubseteq_\infty$ à $\mathcal{D}^0$. Un couple $(g,u) \in \mathcal{D}^0$ est plus grand que $(f,t) \in \mathcal{D}^0$ si et seulement si g est plus précise que f et u est plus grand que t :

$$(f,t) \sqsubseteq^0 (g,u) \Leftrightarrow f \sqsubseteq_\infty g \text{ et } t \leq u \ . \tag{5.8}$$

L'ordre $\sqsubseteq^0$ peut être vu comme une restriction de l'ordre $\sqsubseteq_\infty$: on impose que le support de g (c'est-à-dire le domaine où g est significative) soit plus grand que celui de f. Nous étudions maintenant les propriétés du domaine $\mathcal{D}^0$ muni de l'ordre $\sqsubseteq^0$.

Proposition 5.2 $\left(\mathcal{D}^0, \sqsubseteq^0\right)$ *est un CPO.*

Preuve Le domaine $\left(\mathcal{IF}^0_\infty, \sqsubseteq_\infty\right)$ est un CPO et $\left(\overline{\mathbb{R}_+}, \leq\right)$ muni de l'ordre naturel sur les nombres réels aussi. Le domaine $\mathcal{D}^0$ est le produit cartésien de deux CPOs, c'est donc un CPO. □

Définition 5.11 (Plus petit majorant et plus grand minorant) Soit $(f,t),(g,u) \in \mathcal{D}^0$. On définit le plus petit majorant et le plus grand minorant par :

$$(f,t) \sqcup (g,u) = (\phi, v) \text{ tel que } v = \max(t,u) \text{ et } \phi = \lambda x.f(x) \cap g(x) \ . \tag{5.9}$$

$$(f,t) \sqcap (g,u) = (\phi, v) \text{ tel que } v = \min(t,u) \text{ et } \phi = \lambda x.f(x) \cup g(x) \ . \tag{5.10}$$

Dans ces équations, $\cup$ et $\cap$ sont les opérateurs d'union et d'intersection sur $I_\mathbb{R}$.

Exemple 5.6 La figure 5.7 montre l'effet le l'opérateur $\sqcup$ sur deux fonctions intervalles continues, la figure 5.8 montre l'effet de $\sqcap$. Sur ces deux figures, un élément $(f,t) \in \mathcal{D}^0$ est représenté graphiquement par une représentation de la fonction f comme sur la figure 5.5 et t est représenté par la droite verticale en pointillé.

Nous ajoutons à $\mathcal{IF}^0_\infty$ un plus petit élément $\bot_\infty$ et un plus grand $\top_\infty$. Intuitivement, $\bot_\infty$ correspond à la fonction dont la valeur est $]-\infty,+\infty[$. L'élément $\top_\infty$ est une valeur artificielle qui est plus grande que toutes les fonctions continues réelles. Notons en effet que pour deux fonctions continues réelles f,g que nous voyons comme des éléments de $\mathcal{IF}^0_\infty$, on a $f \sqsubseteq_\infty g \Leftrightarrow f = g$. Comme nous le faisons dans le cas des treillis plats, nous ajoutons donc $\top_\infty$ tel que $\forall f \in \mathcal{C}^0(\mathbb{R}),\ f \sqsubseteq_\infty \top_\infty$. Clairement, on a alors $\forall f \in \mathcal{IF}^0_\infty,\ f \sqsubseteq_\infty \top_\infty$. Le plus grand élément de $\mathcal{D}^0$ est donc $\top^0 = \left(\top_\infty, +\infty\right)$, et le plus petit est $\bot^0 = \left(\bot_\infty, 0\right)$.

Remarque Dans la définition du plus petit majorant de (f,t) et (g,u) (définition 5.11), on suppose que $\forall x \in \mathbb{R}_+,\ f(x) \cap g(x) \neq \emptyset$. Si ce n'est pas le cas, on posera $f \sqcup g = \top^0$.

Proposition 5.3 $(\mathcal{D}^0, \sqsubseteq^0, \top^0, \bot^0, \sqcup, \sqcap)$ *est un treillis complet.*

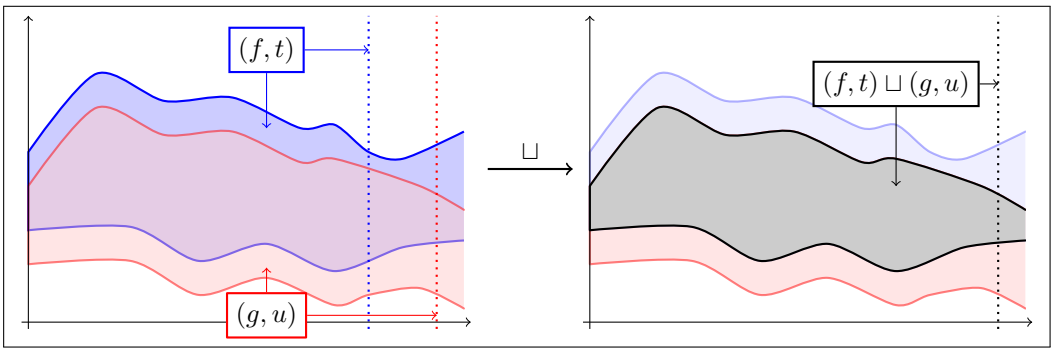


FIG. 5.7 – Union sur $\mathcal{D}^0$.

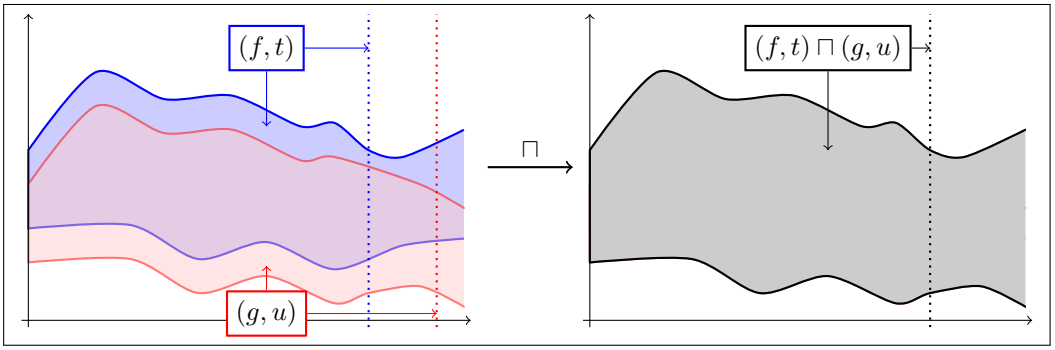


FIG. 5.8 – Intersection sur $\mathcal{D}^0$.

Preuve Comme $(\overline{\mathbb{R}_+}, \leq)$ est un treillis complet, il suffit de montrer que $(\mathcal{IF}^0_\infty, \sqsubseteq_\infty)$ est un treillis complet pour arriver au résultat. Soit donc $S \subseteq \mathcal{IF}^0_\infty$. Soient $\underline{S}$ et $\overline{S}$ les ensembles de fonctions réelles définis par $\underline{S} = \{\underline{f} \ : \ f \in S\}$ et $\overline{S} = \{\overline{f} \ : \ f \in S\}$. L'ensemble $\underline{S}$ est un ensemble de fonction semi-continues inférieurement, donc la fonction $\underline{\mathfrak{f}}$ définie par $\forall x \in \mathbb{R}_+,\ \underline{\mathfrak{f}}(x) = \sup\{\underline{f}(x) \ : \ \underline{f} \in \underline{S}\}$ est semi-continue inférieurement. De même, la fonction $\overline{\mathfrak{f}}(x) = \inf\{\overline{f}(x) \ : \ \overline{f} \in \overline{S}\}$ est semi-continue supérieurement. On définit alors $\mathfrak{f} \in \mathcal{IF}^0_\infty$ par :

$$\mathfrak{f} = \begin{cases} \top_\infty & \text{si } \exists x \in \mathbb{R}_+ \ : \ \underline{\mathfrak{f}}(x) > \overline{\mathfrak{f}}(x) \\ \lambda x.[\underline{\mathfrak{f}}(x), \overline{\mathfrak{f}}(x)] & \text{sinon} \end{cases} .$$

On a bien $\mathfrak{f} = \bigsqcup S$ et $\mathfrak{f} \in \mathcal{IF}^0_\infty$, ce qui prouve que $\mathcal{IF}^0_\infty$ est un treillis complet.

□

5.3.2 Opérations sur $\mathcal{D}^0$

Le treillis $\mathcal{D}^0$ est un ensemble de fonctions qui servent à approcher des fonctions réelles définies sur $\mathbb{R}_+$. Une fonction $f \in \mathcal{IF}^0_\infty$ est en effet un élément à information partielle, et nous allons lui ajouter de l'information en nous servant de l'équation différentielle de la définition 5.4 et d'un opérateur de Picard modifié. Pour cela, nous étendons les opérations classiques sur les fonctions réelles aux éléments de $\mathcal{D}^0$. Nous commençons par les opérateurs arithmétiques, qui sont des extensions point à point des opérateurs sur les intervalles. Pour tout $\odot \in \{+, -, \times, /\}$, on définit :

$$\begin{gathered} \forall (f,t),(g,t) \in \mathcal{D}^0,\ (f,t) \odot (g,t) = (h,t) \in \mathcal{D}^0 \text{ tel que} \\ \forall x \in \mathbb{R}_+,\ h(x) = f(x) \odot g(x) = \{y \odot z \ : \ y \in f(x) \text{ et } z \in g(x)\} \ . \end{gathered} \tag{5.11}$$

Nous définissons maintenant la composition d'une fonction $y \in \mathcal{IF}^0_\infty$ avec une fonction continue par morceaux $F \in \mathcal{C}^0_m(\mathbb{R})$. La difficulté vient des points de discontinuité de la fonction F

où potentiellement les fonctions supérieures et inférieures peuvent perdre la propriété de semi-continuité. Pour cela, on définit pour tout $x_0 \in \mathbb{R}_+$ la valeur de $F \circ y(x_0)$ comme l'union des limites à gauche et à droite de $F \circ y$ en x_0. Rappelons que pour une fonction $\phi : \mathbb{R}_+ \to D$ à valeur dans un espace métrique D, la limite à gauche de f en x est :

$$\lim_{x' \to x^-} f(x') = \lim_{\substack{x' \to x \\ x' < x}} f(x') \ .$$

La limite à droite $\lim_{x' \to x^+} f(x')$ est définie de manière équivalente, il suffit de changer le sens de l'inégalité.

Définition 5.12 (Composition de fonctions) Soit $f \in \mathcal{IF}^0_\infty$ et $F \in \mathcal{C}^0_m(\mathbb{R})$. On définit la composition de f et F comme la fonction $g = F \circ f \in \mathcal{IF}^0_\infty$ par :

$$\forall x \in \mathbb{R}_+, (F \circ f)(x) = \big(\lim_{x' \to x^-} \{F(y, x') \ : \ y \in f(x')\} \big) \sqcup \big(\big(\lim_{x' \to x^+} \{F(y, x') \ : \ y \in f(x')\} \big) \ .$$

Pour un élément $(f, t) \in \mathcal{D}^0$ et $F \in \mathcal{C}^0(\mathbb{R})$, on définit la composition par $F \circ (f, t) = \big(F \circ f, t\big)$.

Remarque Pour $f \in \mathcal{IF}^0_\infty$ et $F \in \mathcal{C}^0_m(\mathbb{R})$, $F \circ f \in \mathcal{IF}^0_\infty$: $\forall x \in \mathbb{R}_+$, $f(x)$ est un intervalle, donc par continuité de F en y, $\{F(y, x) \ : y \in f(x)\}$ est un intervalle et les fonctions supérieures et inférieures sont semi-continues supérieurement et inférieurement.

Exemple 5.7 Soit $f \in \mathcal{IF}^0_\infty$ définie par $\forall x \in \mathbb{R}_+, \ f(x) = [0, x]$. Soit $F \in \mathcal{C}^0_m(\mathbb{R})$ définie par :

$$\forall y, x \in \mathbb{R} \times \mathbb{R}_+, \ F(y, x) = \begin{cases} 5 - 0.5y & \text{si } x < 3 \\ -0.5y & \text{sinon} \end{cases}$$

Alors, $F \circ f \in \mathcal{IF}^0_\infty$ est définie par :

$$\forall x \in \mathbb{R}_+, \ F \circ f(x) = \begin{cases} \left[5 - \frac{x}{2}, 5\right] & \text{si } x < 3 \\ \left[-\frac{3}{2}, 5\right] & \text{si } x = 3 \\ \left[-\frac{x}{2}, 0\right] & \text{sinon} \end{cases}$$

La figure 5.9 montre les fonctions f et $F \circ f$.

Définition 5.13 (Primitive) On définit pour tout $f \in \mathcal{IF}^0_\infty$ et tout $a \leq b \in \mathbb{R}_+$ l'intégrale de f entre a et b par :

$$\int_a^b f(s)ds = \left[\int_a^b \underline{f}(s)ds, \int_a^b \overline{f}(s)ds \right]$$

On définit alors l'opérateur $I : \mathcal{D}^0 \to \mathcal{D}^0$ par $I(f, t) = (g, t)$ avec $\forall x \in \mathbb{R}_+, \ g(x) = \int_0^x f(s)ds$. Remarquons que la propriété de Chasles est vérifiée : $\forall f \in \mathcal{IF}^0_\infty$ et $a \leq b \leq c \in \mathbb{R}_+$, $\int_a^c f(s)ds = \int_a^b f(s)ds + \int_b^c f(s)ds$.

Remarque (1) Pour toute fonction $f \in \mathcal{IF}^0_\infty$, les fonction $\underline{f}$ et $\overline{f}$ sont semi-continues inférieurement et supérieurement, elles sont donc mesurables et bornées, et donc intégrables.

Remarque (2) Pour tout $f \in \mathcal{IF}^0_\infty$, on a bien $I(f) \in \mathcal{IF}^0_\infty$ grâce à la continuité de l'opérateur $\int$ sur les fonctions réelles.

Exemple 5.8 Soit $g = F \circ f \in \mathcal{IF}^0_\infty$ la fonction définie à l'exemple 5.7. Alors, $I(g) \in \mathcal{IF}^0_\infty$ est définie par :

$$\forall x \in \mathbb{R}_+, \ I(g)(x) = \begin{cases} \left[5x - \frac{x^2}{4}, 5x\right] & \text{si } x \leq 3 \\ \left[15 - \frac{x^2}{4}, 15\right] & \text{sinon} \end{cases}$$

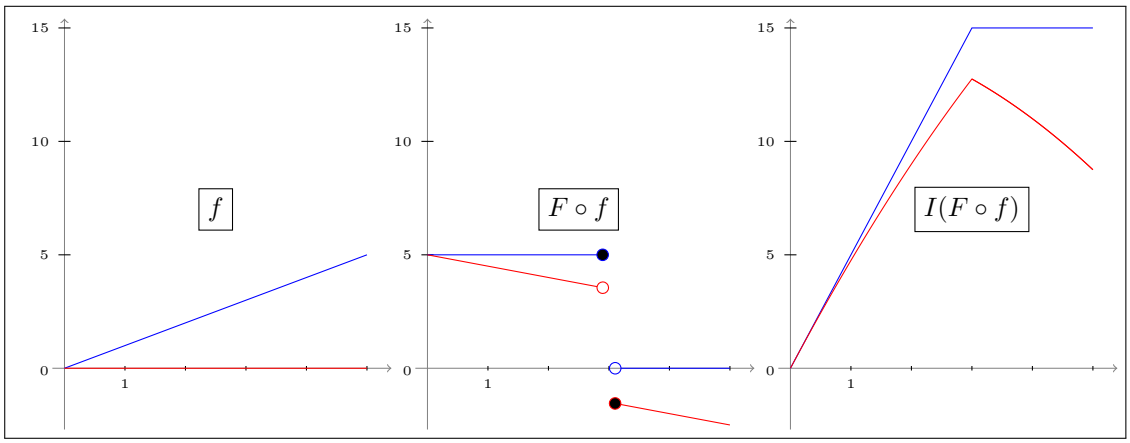


FIG. 5.9 – Exemples de composition d'une fonction continue intervalle avec une fonction continue par morceaux et de calcul de la primitive. La fonction bleue est la fonction supérieure, la fonction rouge est la fonction inférieure.

Définition 5.14 (Largeur) On définit maintenant la largeur d'un élément $(f,t) \in \mathcal{D}^0$ comme la largeur maximale de f sur l'intervalle $[0,t]$, c'est-à-dire l'intervalle où f est significative. On définit donc la fonction $w : \mathcal{D}^0 \to \mathbb{R}_+$ par :

$$\forall (f,t) \in \mathcal{D}^0,\ w(f,t) = \sup_{x \in [0,t]} w(f(x)) \ .$$

Rappelons que pour un intervalle $y = \left[\underline{y}, \overline{y}\right] \in I_{\mathbb{R}}$, $w(y)$ est la largeur de l'intervalle, c'est-à-dire $w = |\overline{y} - \underline{y}|$.

Exemple 5.9 Soit F et f les fonctions définies à l'exemple 5.7. On a $w(f,3) = 3$ et $w(F \circ f, 5) = 6.5$.

Proposition 5.4 *Pour toute fonction continue $F \in \mathcal{C}^0_m(\mathbb{R})$, la fonction $\lambda f.F \circ f$ est monotone et continue. De même, pour tout $t \in \overline{\mathbb{R}_+}$, la fonction $\lambda f.w(f,t)$ est une fonction monotone et continue de $(\mathcal{D}^0, \sqsubseteq^0)$ dans $(\overline{\mathbb{R}_+}, \preceq)$ avec $x \preceq y \Leftrightarrow y \leq x$ et la fonction $I : \mathcal{D}^0 \to \mathcal{D}^0$ est monotone et continue.*

5.3.3 Calcul de la sémantique

Nous allons maintenant définir la sémantique d'une équation différentielle comme la limite d'une suite de fonctions intervalles continues. Cette suite de fonctions d'approximation est construite comme les itérées d'un opérateur de Picard modifié. Cet opérateur Γ travaillera sur le domaine $\mathcal{D}^0$, et modifiera un couple (f,t) de deux façons : d'une part pour tout $x \in [0,t]$, la valeur est transformée en une valeur plus proche de $y_\infty(x)$, et d'autre part la fonction f est redéfinie pour $x \geq t$. Pour la première étape, on utilise l'opérateur de Picard (équation (3.3)), dont nous rappelons ici la définition :

$$P_{[0,X]}(F, y_0)(f) = \lambda x.y_0 + \int_0^x F\big(f(s), s\big) ds \ .$$

Pour la seconde étape, nous étendons la fonction f de telle sorte que, si $f(t)$ contient $y_\infty(t)$, alors $f(x)$ contient $y_\infty(x)$ pour tout $x \geq t$. Pour cela, nous utilisons le fait que la fonction F de l'équation (5.2) a une constante de Lipschitz α. Ainsi, on peut appliquer la proposition 3.4 qui encadre la solution d'un problème de Cauchy sur tout son domaine de définition.

Définition 5.15 (Opérateur de Picard modifié) Soit $F \in \mathcal{C}^0_m(\mathbb{R})$ une fonction globalement α-Lipschitz en y. Pour tout $y_0 \in \mathbb{R}$, on définit la fonction $\Gamma_{F,y_0} : \mathcal{D}^0 \to \mathcal{D}^0$ par :

$$\forall (f,t) \in \mathcal{D}^0,\ \Gamma_{F,y_0}(f,t) = (g, t+1) \text{ avec } g = \lambda x. \begin{cases} P_{[0,t]}\big(F, y_0\big)(f)(x) & \text{si } x \leq t \\ J + F(J) \cdot [-e^{\alpha(x-t)}, e^{\alpha(x-t)}] \cdot (x-t), & \\ \quad \text{avec } J = P_{[0,t]}\big(F, y_0\big)(f)(t) & \text{sinon} \end{cases}$$

La formule ci-dessus est valide pour $f \neq \bot$ et $t > 0$. On définit donc les cas particuliers :

$$\forall t \in \mathbb{R}_+,\ \Gamma_{F,y_0}(\bot, t) = \lambda x.[y_0, y_0] + F(y_0) \cdot x \cdot [-e^{\alpha x}, e^{\alpha x}] \text{ et } \forall y \in \mathcal{D}^0,\ \Gamma_{F,y_0}(y, 0) = y\ .$$

Remarque L'opérateur de Picard rend donc une approximation meilleure (quand $x \leq t$ dans la définition 5.15) et rajoute de l'information vers la droite : le support passe de t à $t+1$. Autrement dit, si on pose $(g, t+1) = \Gamma_{F,y_0}(f,t)$, la fonction g est non seulement plus précise que f mais également significative sur un plus grand domaine. Le choix d'augmenter le support t de 1 est arbitraire, on aurait pu choisir n'importe quel réel strictement positif.

Exemple 5.10 La figure 5.10 montre l'effet de Γ_{F,y_0} sur des fonctions intervalles. La ligne noire représente y_∞. Sur la figure du haut, on voit comment l'information portée par la fonction bleue est améliorée (cas où $x \leq t$ dans la définition 5.15) ; la figure du bas montre comment on rajoute de l'information en étendant la fonction rouge sur $[t, +\infty[$ (cas où $x \geq t$ dans la définition 5.15).

Proposition 5.5 (Monotonie de Γ_{F,y_0}) *L'opérateur $\Gamma_{F,y_0} : \mathcal{D}^0 \to \mathcal{D}^0$ est une fonction monotone.*

Preuve Soit $(f,t) \in \mathcal{D}^0$ et $(f',t') \in \mathcal{D}^0$ tels que $(f,t) \sqsubseteq^0 (f',t')$. On a donc :

$$t \leq t' \text{ et } \forall x \in \mathbb{R}_+,\ f'(x) \subseteq f(x)\ .$$

Posons $(g,u) = \Gamma_{F,y_0}(f,t)$ et $(g',u') = \Gamma_{F,y_0}(f',t')$. On veut montrer que $(g,u) \sqsubseteq^0 (g',u')$.
On a $u = t+1$ et $u' = t'+1$, donc $u \leq u'$. Montrons maintenant que $\forall x \in \mathbb{R}_+,\ g'(x) \subseteq g(x)$.
Soit donc $x \in \mathbb{R}_+$. Supposons que $x \in [0,t]$. On a alors $g(x) = y_0 + I\big(F \circ f\big)(x)$ et $g'(x) = y_0 + I\big(F \circ f'\big)(x)$. D'après la monotonie des opérateurs I et $\circ$ (proposition 5.4), on a $I\big(F \circ f\big) \sqsubseteq_\infty I\big(F \circ f'\big)$ et donc

$$I\big(F \circ f'\big)(x) \subseteq I\big(F \circ f\big)(x)$$

d'où $g'(x) \subseteq g(x)$.
Supposons maintenant que $x \in [t,u]$. On a alors $g(x) = J + F(J) \cdot [-e^{\alpha(x-t)}, e^{\alpha(x,t)}] \cdot (x-t)$ avec $J = y_0 + \int_0^x F(f(s), s)ds$. On a également $g'(x) = y_0 + \int_0^t F(f'(s), s)ds + \int_t^x F(f'(s), s)ds$. Clairement, $y_0 + \int_0^t F(f'(s), s)ds \subseteq J$, il reste à montrer que

$$\int_t^x F(f'(s), s)ds \subseteq F(J) \cdot [-e^{\alpha(x-t)}, e^{\alpha(x-t)}] \cdot (x-t)\ .$$

Cet encadrement est vrai en utilisant la propriété 3.4 des équations différentielles.
Soit enfin $t \geq u$. La preuve que $g'(x) \subseteq g(x)$ repose également sur la propriété 3.4. □

Théorème 5.6 *L'opérateur Γ_{F,y_0} admet un plus petit point fixe (y,τ) tel que :*

$$(y,\tau) = \bigsqcup_{n \in \mathbb{N}} \Gamma^n_{F,y_0}(\bot^0)\ .$$

De plus, $\tau = +\infty$ et $y = y_\infty$, où y_∞ est la solution de l'équation différentielle de la définition 5.4.

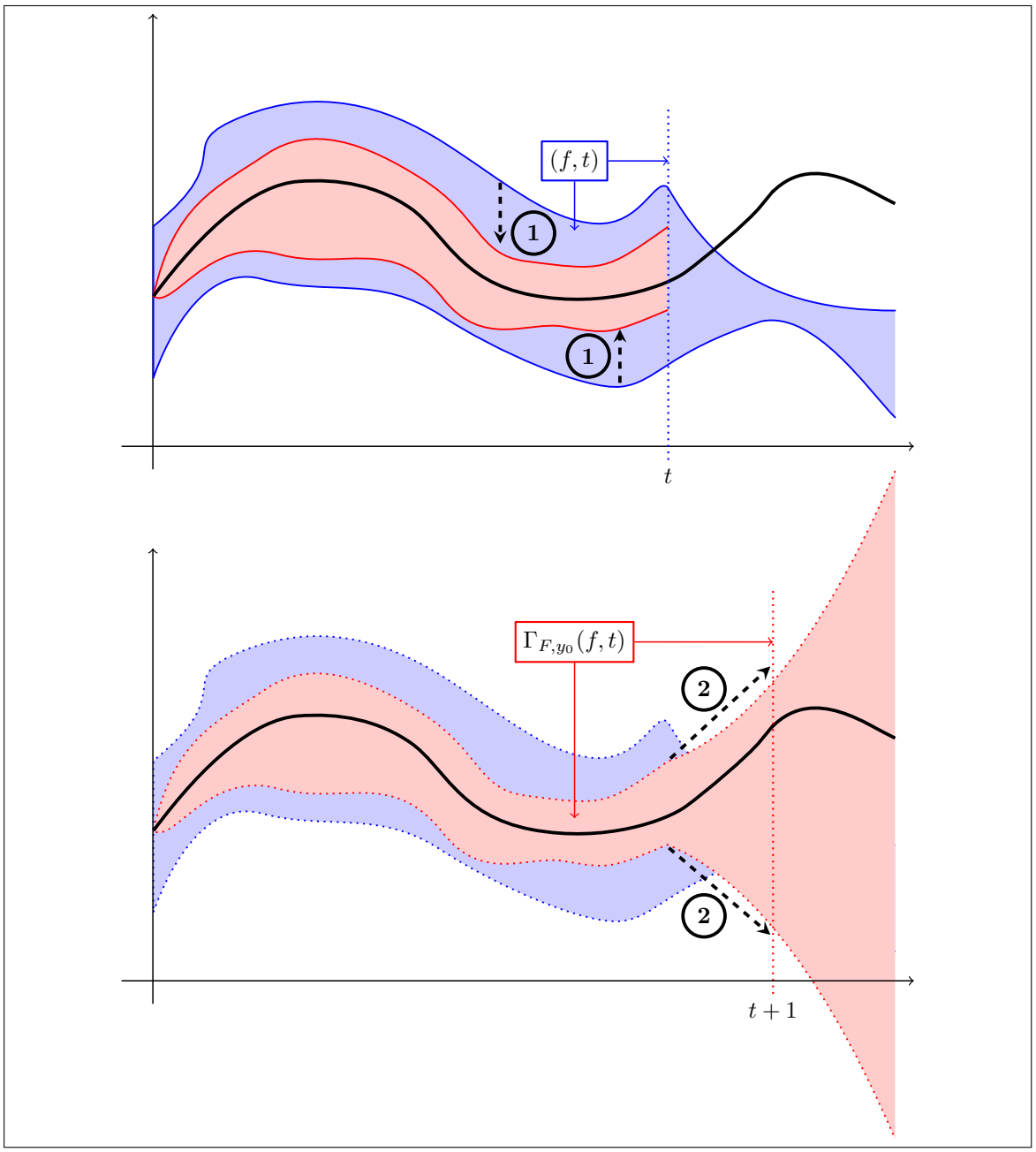


FIG. 5.10 – Les deux étapes de la fonction Γ_{F,y_0}.

Preuve Nous prouvons dans un premier temps l'existence du plus petit point fixe. Nous avons vu que Γ_{F,y_0} est un opérateur monotone sur $\mathcal{D}^0$. Comme $\mathcal{D}^0$ est un CPO (proposition 5.2), d'après le théorème de Kleene (théorème 2.3) Γ_{F,y_0} possède un plus petit point fixe qui se calcule comme la limite des itérées de Γ_{F,y_0} en partant de $\perp^0$. Il nous reste alors à prouver que la solution y_∞ de l'équation différentielle vérifie $(y_\infty, +\infty) = \bigsqcup_{n\in\mathbb{N}} \Gamma^n(\perp_\infty, 0)$. Pour cela, nous aurons besoin d'un lemme supplémentaire.

Lemme 5.7 *Soit $(f,t) \in \mathcal{D}^0$, et soit $(g,u) = \Gamma_{F,y_0}(f,t)$. On a alors :*

$$\forall x \in [0,t], y_\infty(x) \in f(x) \Rightarrow \forall x \in \mathbb{R}_+,\ y_\infty(x) \in g(x)$$

Preuve Soit $(f,t) \in \mathcal{D}^0$ tel que $\forall x \in [0,t]\,,\ y_\infty(x) \in f(x)$. On note $(g,u) = \Gamma(f,t)$. Soit maintenant $x \in \mathbb{R}_+$, montrons que $y_\infty(x) \in g(x)$.

- si $x \in [0,t]$, on a $\forall s \in [0,x]\,,\ y_\infty(s) \in f(s)$, donc $F\big(y_\infty(s)\big) \in F\big(f(s)\big)$, et donc $y_0 + \int_0^x F\big(y_\infty(s)\big)ds \in y_0 + \int_0^x F\big(f(s)\big)ds$. Comme y_∞ est le point fixe de l'opérateur de Picard $P_{[0,X_f]}\big(F, y_0\big)$, on a $y_0 + \int_0^x F\big(y_\infty(s)\big)ds = y_\infty(x)$, et donc $y_\infty(x) \in g(x)$.
- si $x > t$, on sait que $y_\infty(t) \in f(t)$. On peut alors utiliser l'encadrement classique de la solution d'une EDO $\dot{y} = F(y)$, où F a une constante de Lipschitz k (proposition 3.4) :

  $$\forall t \geq t_0,\ \|y(t) - y(t_0)\| \leq e^{k.|t-t_0|}.F(y(t_0)).(t-t_0)$$

  Dans notre cas, , on a $y_\infty(t) \in J = P_{[0,t]}\big(F, y_0\big)(f)(t)$ et $x \geq t$, donc $y_\infty(x) \in y_\infty(t) + [-e^{\alpha(x-t)}, e^{\alpha(x-t)}].F(y_\infty(t)).(x-t)$, soit : $y_\infty(x) \in J + [-e^{\alpha(x-t)}, e^{\alpha(x-t)}].F(J).(t-X)$, c'est-à-dire : $y_\infty(x) \in g(x)$.

□

Preuve (Preuve du théorème 5.6) Posons pour tout $n \in \mathbb{N}$ $(f_n, t_n) = \Gamma^n_{F,y_0}(\perp, 0)$. D'après la définition de Γ_{F,y_0}, on sait que $t_1 = 1$ et f_1 vérifie $\forall x \in \mathbb{R}_+,\ y_\infty(x) \in f_1(x)$. On montre alors par récurrence sur n que $\forall n \geq 1,\ t_n = n$, et en utilisant le lemme 5.7 on montre que $\forall x \in \mathbb{R}_+,\ y_\infty(x) \in f_n(x)$. On en déduit alors que $\forall n \in \mathbb{N},\ (f_n, t_n) \sqsubseteq^0 (y_\infty, +\infty)$.
Soit maintenant $(f,\tau) = \bigsqcup_{n\in\mathbb{N}}(f_n, t_n)$. D'après la définition du plus petit majorant, on a clairement $(f,\tau) \sqsubseteq^0 (y_\infty, +\infty)$. Remarquons que $\forall n \in \mathbb{N},\ \tau \geq t_n$, de sorte que $\tau = +\infty$. Montrons maintenant que $f = y_\infty$.
D'après le théorème de Picard (théorème 3.2) et la définition de Γ_{F,y_0} (définition 5.15), on a pour tout $x \in \mathbb{R}_+$, $\lim_{n\to\infty} w(f_n(x)) = 0$. Comme pour tout $x \in \mathbb{R}_+$ et pour tout $n \in \mathbb{N}$, $f(x) \in f_n(x)$, on a clairement $\forall x \in \mathbb{R}_+,\ w(f(x)) = 0$. Donc, f est une fonction continue réelle (c'est-à-dire $f \in \mathcal{C}^0(\mathbb{R}_+)$) vérifiant $f \sqsubseteq_\infty y_\infty$. D'après la définition de l'ordre $\sqsubseteq_\infty$ (définition 5.6), cela impose que $f = y_\infty$. □

On peut maintenant donner la définition formelle de la sémantique d'une équation différentielle.

Définition 5.16 (Sémantique d'une équation différentielle) Étant donnée une fonction continue par morceaux $F \in \mathcal{C}^0_m(\mathbb{R})$ et $y_0 \in \mathbb{R}$, soit $(y_\infty, +\infty) = \bigsqcup_{n\in\mathbb{N}} \Gamma^n_{F,y_0}(\perp_\infty, 0)$. On pose alors $[\![F, y_0]\!]^c = y_\infty$.

Exemple 5.11 La figure 5.11 présente les premières itérées du calcul de la sémantique du modèle continue $\kappa = \big(Y, 1, \{F_0, F_1\}, (y_0, k_0)\big)$ avec $\forall y \in \mathbb{R},\ F_0(y) = y$ et $F_1(y) = -y$ sous l'action de la fonction $s : \mathbb{R}_+ \to \mathbb{B}$ définie par $\forall t \in \mathbb{R}_+,\ s(t) = \begin{cases} 0 & \text{si } t \leq 1 \\ 1 & \text{sinon} \end{cases}$.
La courbe noire représente la sémantique $[\![\kappa]\!]_s$, c'est-à-dire la solution de l'équation différentielle donnée à la définition 5.4.

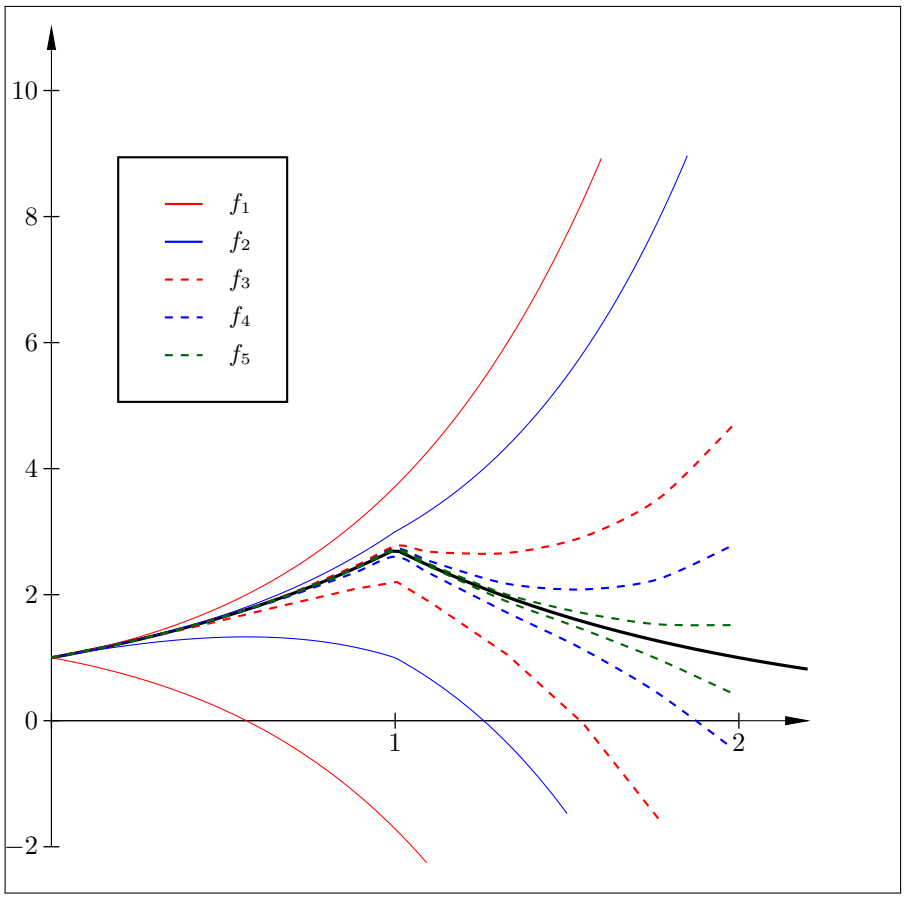


FIG. 5.11 – Exemple d'itérations de Γ_{F,y_0}.

5.4 Sémantique dénotationnelle de la partie discrète

Nous donnons maintenant une sémantique dénotationnelle pour les sous-parties discrètes des systèmes hybrides, en supposant que l'on peut connaître exactement l'évolution de la partie continue. On s'intéresse à un modèle discret $\Delta = \big(CVar, m, P\big)$, composé d'un ensemble de variables continues et d'un programme écrit en **H-SIMPLE**. Pour simplifier la compréhension de cette section, nous ferons encore l'hypothèse que $CVar$ ne possède qu'une seule variable continue. La sémantique de Δ est la sémantique du programme P soumis à un environnement extérieur κ. On va donc définir une sémantique dénotationnelle paramétrée par l'environnement κ comme une fonction $[\![P]\!]_\kappa$ entre états étendus. Nous commençons donc par définir cette notion d'état étendu, puis donnons les dénotations de chaque instruction du langage **H-SIMPLE**.

5.4.1 États étendus

Comme nous l'avons vu au chapitre 2, un état discret du programme P est une fonction partielle associant à chaque variable discrète $v \in Var$ un nombre à virgule flottante $\boldsymbol{f} \in \mathbb{F}$. Cependant, un programme écrit en **H-SIMPLE** agit sur d'autres éléments que les variables discrètes. En effet, P peut modifier le temps d'exécution via une instruction ***wait*** et l'environnement continu via une instruction ***act***. Nous devons donc étendre la notion d'états pour prendre en compte à la fois l'environnement continu et le temps d'exécution. Nous supposons donc que l'on dispose de trois variables $t, act, y \notin Var$. La variable t représente le temps d'exécution et sera donc un nombre réel positif. La variable y représente l'environnement extérieur et sera donc associée à une fonction continue du temps. Enfin, la variable act représente la suite des commandes envoyées par le programme aux actionneurs au cours de son exécution, sa valeur sera donc une fonction constante par morceaux à valeur dans $\mathbb{B}^m$. Cette fonction est celle que nous supposions connue pour définir la sémantique de la partie continue (section 5.3).

Définition 5.17 (État étendu) L'état étendu d'un modèle discret $\Delta = \big(CVar, m, P\big)$ est une fonction σ associant à chaque variable discrète un nombre à virgule flottante, à t un nombre réel positif, à y une fonction continue à valeurs réelles et à act une fonction constante par morceaux à valeurs dans $\mathbb{B}^m$. Nous noterons $\mathit{Const}(\mathbb{B}^m)$ l'ensemble des fonctions constantes par morceaux à valeurs dans $\mathbb{B}^m$. Nous noterons Σ_e l'ensemble des états étendus :

$$\Sigma_e = \big\{\big(Var \to Val\big) \times \big(\{t\} \to \mathbb{R}_+\big) \times \big(\{act\} \to \mathit{Const}(\mathbb{B}^m)\big) \times \big(\{y\} \to \mathcal{C}^0(\mathbb{R}_+)\big)\big\}\,. \tag{5.12}$$

Pour tout état $\sigma \in \Sigma_e$, nous noterons $\sigma(X) \in \mathbb{F}$ la valeur d'une variable discrète $X \in Var$, et pour toute variable $v \in \{t, act, y\}$, nous noterons $\sigma.v$ la valeur prise par v dans σ.

Remarque Avec les notations de la définition 5.17, la valeur instantanée de l'environnement continu est donnée par $\sigma.y(\sigma.t)$. Comme à la section 2.2.2, nous noterons $\sigma[V \mapsto v]$ l'environnement égal à σ sauf pour la variable $V \in Var \cup \{t, act, y\}$ qui prend la valeur v. Pour tout $v \in \mathbb{B}^m$, $k \in [1, m]$ et $c \in \{0, 1\}$, nous noterons de même $v[k \mapsto c]$ le vecteur de booléens égal à v sauf pour la k-ième coordonnée qui vaut c.

5.4.2 Dénotations des instructions discrètes

Les dénotations associées aux instructions discrètes (**Stmt** de la figure 5.2) sont exactement les mêmes que pour les instructions du langage **SIMPLE** définies à la figure 2.4.

5.4.3 Dénotations des instructions hybrides

Les dénotations des instructions hybrides de **HStmt** sont des fonctions modifiant un environnement étendu : $\forall hs \in \mathbf{HStmt}$, $[\![hs]\!] : \Sigma_e \to \Sigma_e$. Ces fonctions, données par la figure 5.12, fonctionnent ainsi :

- pour tout $X \in Var$ et $y \in CVar$, $\boldsymbol{sens}.y?X$ modifie la variable X et lui associe le nombre flottant le plus proche de la valeur de la variable continue y à l'instant ou l'action est exécutée (équation (5.13)). L'opérateur $\uparrow_\sim$ de l'équation (5.13) effectue l'opération de transtypage.
- pour tout $u \in \mathbb{R}_+$, $\boldsymbol{wait}\ u$ modifie la valeur de la variable t et lui donne la valeur $t + u$ (équation (5.14)).
- pour tout $k \in [0, m]$ et $c \in \{0, 1\}$, $\boldsymbol{act}.k!c$ modifie un environnement σ comme suit. On rajoute à la fonction $\sigma.act$ une marche dont la valeur est la dernière valeur prise par $\sigma.act$ sauf que la k-ième coordonnée est changée en c (on obtient l'état σ_i dans l'équation (5.15)). Ensuite, on change la valeur de la fonction continue y en la solution de l'équation différentielle donnée par la nouvelle fonction $\sigma_i.act$, c'est-à-dire en la sémantique $[\![\kappa]\!]_{\sigma_i.act}$.

$$[\![\boldsymbol{sens}.\mathrm{y}\ ?\mathrm{X}]\!]_\kappa = \left\{(\sigma, \sigma')\ :\ \sigma' = \sigma[x \mapsto \uparrow_\sim (\sigma.y(\sigma.t))]\right\} \tag{5.13}$$

$$[\![\boldsymbol{wait}\ u]\!]_\kappa = \left\{(\sigma, \sigma')\ :\ \sigma' = \sigma[t \mapsto \sigma.t + u]\right\} \tag{5.14}$$

$$[\![\boldsymbol{act}.k!c]\!]_\kappa = \left\{ (\sigma, \sigma')\ :\ \exists \sigma_i \in \Sigma_e, \begin{array}{l} \sigma_i = \sigma\left[act \mapsto \lambda x. \left\{ \begin{array}{ll} \sigma.act(x) & \text{si } x \leq \sigma.t \\ \sigma.act(\sigma.t)[k \mapsto c] & \text{sinon} \end{array} \right.\right] \\ \sigma' = \sigma_i\left[y \mapsto [\![\kappa]\!]_{\sigma_i.act}\right] \end{array} \right\} \tag{5.15}$$

FIG. 5.12 – Dénotations pour les instructions hybrides.

Exemple 5.12 On reprend et modifie le programme de l'exemple 2.8 de la section 2.2.3. Soit donc le programme **H-IMP** suivant :

$P ::=$ $i = 1; h = 0.01; res = 0;$
$\boldsymbol{while}\ i \leq 100\ \boldsymbol{do}$
$\boldsymbol{sens}.y?X;$
$res = res + X \times h;$
$i = i + 1;$
$\boldsymbol{wait}\ 0.01;$

Le programme P calcule, par la méthode des rectangles, l'intégrale des données qu'il reçoit via les capteurs. Si on considère le modèle continu $\kappa = \left(y, 0, \{\lambda x.x\}\right)$, la sémantique de P sous l'environnement κ est donné par : $\forall \sigma \in \Sigma_e,\ [\![P]\!]_\kappa(\sigma) = \sigma[i \mapsto 100][t \mapsto 1][X \mapsto \sum_{i=1}^{100} e^{0.01*i} * 0.01]$.

5.5 Sémantique dénotationnelle hybride

Dans la sémantique discrète définie à la section 5.4, nous supposons qu'à chaque instruction $\boldsymbol{act}$, on peut calculer la sémantique de l'environnement continu, c'est-à-dire la solution d'une équation différentielle comme défini en section 5.3. Cependant, nous avons vu que ce calcul nécessite une itération infinie de l'opérateur de Picard modifié (définition 5.15). Nous allons donc essayer de ne pas calculer ce point fixe à chaque instruction $\boldsymbol{act}$ mais de calculer en parallèle la sémantique discrète et la sémantique continue. Soit $\Omega = \left(\Delta, \kappa\right)$ un système hybride, avec $\Delta = \left(CVar, m, P\right)$ et $\kappa = \left(Var, m', \{F_k\}_{k \in \mathbb{B}^m}, y_0\right)$. Pour construire sa sémantique, nous allons calculer la sémantique du programme P ainsi que l'évolution du milieu continu en parallèle, en respectant les types de communications entre les deux :

- des *données* sont passées, via les capteurs, de κ à Δ (instructions $\boldsymbol{sens}$). Cette communication est bloquante : il faut que le programme et l'environnement continu aient atteint le même temps d'exécution pour que l'échange de données s'exécute.

– des *ordres* sont passés, via les actionneurs, de Δ à κ. En effet, le programme indique seulement via les instructions ***act*** comment l'environnement continu évoluera dans le futur en choisissant une des fonctions F_k comme dynamique du système. Cette communication n'est donc pas bloquante : l'environnement continu ne doit pas forcément avoir atteint le temps d'exécution du programme lorsque l'instruction est exécutée.

La sémantique que nous construisons et notamment les dénotations des instructions ***sens*** et ***act*** respectent ces principes. Nous définissons la sémantique ainsi : nous introduisons d'abord des environnements hybrides (section 5.5.1) qui comportent une approximation du résultat du programme et une approximation de l'évolution continue, puis nous définissons les dénotations de chaque instruction comme des fonctions entre environnements hybrides (sections 5.5.2 et 5.5.3), et enfin nous présentons la sémantique hybride (section 5.5.4).

5.5.1 Environnements hybrides

Un environnement hybride est un couple $(\sigma_\delta, \sigma_\kappa)$ tel que σ_δ est un environnement discret et σ_κ un environnement continu. Les environnements discrets σ_δ représentent l'évolution du programme : ils associent donc à chaque variable discrète un nombre à valeur flottante, et comme pour la sémantique discrète (section 5.4), ils possèdent deux variables supplémentaires : t qui représente le temps d'exécution et act qui représente la valeur de la commande envoyée aux actionneurs. L'environnement σ_δ associe à t un nombre réel positif et à act un vecteur de booléens $b \in \mathbb{B}^m$. On note Σ_Δ l'ensemble des environnements discrets :

$$\Sigma_\Delta = \left\{ (Var \to Val) \times (\{t\} \to \mathbb{R}_+) \times (\{act\} \to \mathbb{B}^m) \right\} . \tag{5.16}$$

Les environnements continus contiennent une approximation des variables continues (c'est-à-dire un élément de $\mathcal{D}^0$) ainsi que la fonction F qui définit la dynamique continue. Cette fonction est une fonction continue par morceaux qui alterne entre toutes les fonctions F_k de κ, et sera progressivement définie par les instructions ***act*** du programme P. On note Σ_κ l'ensemble des environnements continus :

$$\Sigma_\kappa = \left\{ (y \in \mathcal{D}^0) \times (F \in \mathcal{C}^0_m(\mathbb{R})) \right\} . \tag{5.17}$$

On note alors $\Sigma^{\mathcal{H}}$ l'ensemble de tous les environnements hybrides :

$$\Sigma^{\mathcal{H}} = \left\{ (\sigma_\delta, \sigma_\kappa) \ : \ \sigma_\delta \in \Sigma_\Delta, \sigma_\kappa \in \Sigma_\kappa \right\} \tag{5.18}$$

Remarque (Notations) Pour tout $\sigma_\kappa \in \Sigma_\kappa$, $\sigma_\kappa.y \in \mathcal{D}^0$ est donc un couple (f, t) avec $f \in \mathcal{IF}^0_\infty$ et $t \in \mathbb{R}_+$. Par abus de notations on notera pour tout $x \in \mathbb{R}_+$ $\sigma_\kappa.y(x)$ la valeur prise par f en x.

On définit enfin $\Pi_\delta : (\sigma_\delta, \sigma_\kappa) \mapsto \sigma_\delta$ et $\Pi_\kappa : (\sigma_\delta, \sigma_\kappa) \mapsto \sigma_\kappa$ les projections d'un environnement hybride vers les environnements discrets et continus, respectivement.

5.5.2 Dénotations hybrides des instructions discrètes

Les dénotations des instructions discrètes sont des extensions simples des dénotations données au chapitre 2 pour les environnements hybrides : on applique simplement la dénotation discrète à la partie discrète de l'environnement, la partie continue restant inchangée. Les règles donnant ces dénotations hybrides sont détaillées à la figure 5.13.

5.5.3 Dénotations hybrides des instructions hybrides

Les dénotations des instructions hybrides sont des fonctions entre environnements hybrides, données par les équations de la figure 5.14. Pour l'instruction ***wait***, l'environnement continu reste inchangé et seul le temps d'exécution est modifié dans l'environnement discret (équation (5.25)). L'instruction ***sens***$.y?X$ modifie un couple $(\sigma_\delta, \sigma_\kappa)$ ainsi : on commence par améliorer l'approximation continue en appliquant Γ_{F,y_0}. On s'assure en particulier que la fonction intervalle y est significative au temps $\sigma_\delta.t$. On applique donc Γ_{F,y_0} $n = \lceil \sigma_\delta.t \rceil + 1$ fois ; on effectue ainsi un certain

$$
\begin{aligned}
[\![X := a||\kappa]\!]^{\mathcal{H}} &= \big\{\big((\sigma_\delta,\sigma_\kappa),(\sigma'_\delta,\sigma_\kappa)\big) \ : \ (\sigma_\delta,\sigma'_\delta) \in [\![X := a]\!]\big\} && (5.19)\\
[\![s_0; s_1||\kappa]\!]^{\mathcal{H}} &= [\![s_1||\kappa]\!]^{\mathcal{H}} \circ [\![s_0||\kappa]\!]^{\mathcal{H}} && (5.20)\\
[\![\textbf{\textit{if}}\ e\ \textbf{\textit{then}}\ s_0\ \textbf{\textit{else}}\ s_1||\kappa]\!]^{\mathcal{H}} &= \big\{\big(\sigma,\sigma'\big) \ : \ \sigma \in \Sigma^{\mathcal{H}} \text{ et } \exists f \neq 0,\ (\Pi_\delta(\sigma), f) \in [\![e]\!] \wedge \big(\sigma,\sigma'\big) \in [\![s_0||\kappa]\!]^{\mathcal{H}}\big\} && (5.21)\\
&\quad \cup\\
&\quad \big\{\big(\sigma,\sigma'\big) \ : \ \sigma \in \Sigma^{\mathcal{H}} \text{ et } (\Pi_\delta(\sigma), 0) \in [\![e]\!] \wedge \big(\sigma,\sigma'\big) \in [\![s_1||\kappa]\!]^{\mathcal{H}}\big\}\\
[\![\textbf{\textit{while}}\ e\ \textbf{\textit{do}}\ s||\kappa]\!]^{\mathcal{H}} &= lfp(\Gamma) \text{ avec} && (5.22)\\
\Gamma(\varphi) &= \big\{\big(\sigma,\sigma'\big) \ : \ \sigma \in \Sigma^{\mathcal{H}} \text{ et } \exists f \neq 0,\ (\Pi_\delta(\sigma), f) \in [\![e]\!] \wedge \big(\sigma,\sigma'\big) \in \varphi \circ [\![e||\kappa]\!]^{\mathcal{H}}\big\}\\
&\quad \cup\\
&\quad \big\{\big(\sigma,\sigma\big) \ : \ \sigma \in \Sigma^{\mathcal{H}} \text{ et } (\Pi_\delta(\sigma), 0) \in [\![s]\!]\big\}
\end{aligned}
$$

FIG. 5.13 – Dénotations hybrides pour les instructions **Stmt**.

nombre de pas dans le calcul de la sémantique continue. Ensuite, on affecte à X le milieu de la valeur prise par $\sigma_\kappa.y$ à l'instant $\sigma_\delta.t$, c'est-à-dire l'approximation courante de la sémantique continue. Lors de cette affectation, on effectue une approximation de la valeur réelle fournie par la sémantique continue en le nombre à virgule flottante le plus proche (c'est le rôle de la fonction $\uparrow_\sim$ dans l'équation (5.23)). La première étape correspond au calcul de σ'_κ dans l'équation (5.23), la seconde au calcul de σ'_δ. Dans l'équation (5.23), l'entier n est donné par $n = \lceil \sigma_\delta.t \rceil + 1$.
Enfin, l'instruction $\textbf{\textit{act}}.k!c$ modifie un couple $(\sigma_\delta, \sigma_\kappa)$ ainsi : on modifie d'abord dans σ_δ la valeur de act pour que la k-ième coordonnée prenne la valeur c ($\sigma_\delta.act$ devient donc $\sigma_\delta.act[k \mapsto c]$), puis on modifie dans σ_κ la fonction F pour qu'elle suive, à partir de l'instant $\sigma_\delta.t$, les valeurs fournie par la fonction correspondant à l'actionneur choisi. La première étape correspond au calcul de σ'_δ dans l'équation (5.24), la seconde correspond au calcul de σ'_κ.

$$
[\![\textbf{\textit{sens}}.y?X||\kappa]\!]^{\mathcal{H}} = \left\{ \big((\sigma_\delta,\sigma_\kappa),(\sigma'_\delta,\sigma'_\kappa)\big) \ : \ \begin{array}{l} \sigma'_\kappa = \sigma_\kappa[y \mapsto \Gamma^n_{\sigma_\kappa.F,y_0}(\sigma_\kappa.y)] \\ \sigma'_\delta = \sigma_\delta[X \mapsto \uparrow_\sim mid(\sigma'_\kappa.y(\sigma_\delta.t))] \end{array} \right\} \qquad (5.23)
$$

$$
[\![\textbf{\textit{act}}.k!c||\kappa]\!]^{\mathcal{H}} = \left\{ \big((\sigma_\delta,\sigma_\kappa),(\sigma'_\delta,\sigma'_\kappa)\big) \ : \ \begin{array}{l} \sigma'_\delta = \sigma_\delta\big[act \mapsto \sigma_\delta.act[k \mapsto c]\big] \\ \sigma'_\kappa = \sigma_\kappa\left[F \mapsto \lambda x, y. \left\{ \begin{array}{ll} \sigma_\kappa.F(x,y) & \text{si } \ x \leq \sigma_\delta.t \\ F_{\sigma'_\delta.act}(x,y) & \text{sinon} \end{array} \right.\right] \end{array} \right\} \qquad (5.24)
$$

$$
[\![\textbf{\textit{wait}}\ u||\kappa]\!]^{\mathcal{H}} = \big\{\big((\sigma_\delta,\sigma_\kappa),(\sigma'_\delta,\sigma_\kappa)\big) \ : \ \sigma'_\delta = \sigma_\delta[t \mapsto t+u]\big\} \qquad (5.25)
$$

FIG. 5.14 – Dénotations pour les instructions hybrides **HStmt**.

5.5.4 Sémantique hybride

Nous pouvons maintenant définir la sémantique d'un système hybride $\Omega = \big(\Delta, \kappa\big)$. Si $\Delta = \big(CVar, m, P\big)$, on veut poser $[\![\Omega]\!]^{\mathcal{H}} = [\![P||\kappa]\!]^{\mathcal{H}}$. Cependant, avec cette formulation, on ne calcule pas forcément exactement la sémantique continue. En effet, comme nous l'avons vu à la section 5.3, le calcul de la sémantique continue nécessite une itération infinie de l'opérateur de Picard modifié. Dans le calcul de $[\![P||\kappa]\!]^{\mathcal{H}}$, on ne va pas forcément effectuer un nombre infini d'étapes de calcul. Par exemple, si le programme ne possède pas de boucles ***while***, on ne calculera que t itérations de l'opérateur de Picard, où t est le temps d'exécution du programme. On doit donc ajouter un calcul de point fixe qui permet de calculer également l'évolution continue. La sémantique $[\![\Omega]\!]^{\mathcal{H}}$ est

donc une fonction entre environnements hybrides qui modifie un couple $(\sigma_\delta, \sigma_\kappa)$ ainsi : on calcule d'abord $(\sigma'_\delta, \sigma'_\kappa) = [\![P||\kappa]\!]^{\mathcal{H}}(\sigma_\delta, \sigma_\kappa)$, puis deux cas se présentent :

- si $\sigma'_\kappa = \sigma_\kappa$, alors le programme P n'a pas d'effet sur l'environnement continu. Comme seules les instructions ***sens*** modifient σ_κ, cela veut dire que soit il n'y a pas d'instruction ***sens*** dans P, soit celles-ci ne modifient pas σ_κ. Cette dernière hypothèse revient à dire que $\sigma_\kappa.y$ est un point fixe de $\Gamma_{\sigma_\kappa.F, y_0}$, c'est-à-dire que la solution de l'équation différentielle a été atteinte. On peut donc alors considérer que la sémantique de P est la sémantique réelle du programme dans son environnement continu, et on posera donc $[\![\Omega]\!]^{\mathcal{H}}\big(\sigma_\delta, \sigma_\kappa\big) = \big(\sigma'_\delta, \sigma'_\kappa\big)$.
- si $\sigma'_\kappa \neq \sigma_\kappa$, l'environnement continu a été modifié par l'exécution du programme. Cela revient à dire que l'approximation de l'évolution continue dans σ'_κ est meilleure que celle dans σ_κ car on a effectué plusieurs itérations de l'opérateur de Picard. Cependant, l'évolution continue n'est pas encore l'évolution réelle car nous n'avons toujours pas atteint la solution de l'équation différentielle. Par contre, σ'_κ est une meilleur approximation de cette solution, et on va donc recalculer l'évolution du programme en utilisant σ'_κ comme point de départ pour la sémantique continue. On calcule donc $[\![\Delta||\kappa]\!]^{\mathcal{H}}\big(\sigma_\delta, \sigma'_\kappa\big)$.

Définition 5.18 (Sémantique hybride des systèmes hybrides) Soit $\Omega = \big(\Delta, \kappa\big)$ un système hybride avec $\Delta = \big(CVar, m, P\big)$. La sémantique $[\![\Omega]\!]^{\mathcal{H}}$ de Ω est :

$$
\begin{aligned}
[\![\Omega]\!]^{\mathcal{H}} &= Fix(\Gamma^{\mathcal{H}}) \text{ avec} \\
\Gamma^{\mathcal{H}}(\varphi)(\sigma_\delta, \sigma_\kappa) &= (\sigma'_\delta, \sigma'_\kappa) \text{ tel que } \begin{cases} \sigma'_\delta = \Pi_\delta\big([\![\Delta||\kappa]\!]^{\mathcal{H}}(\sigma_\delta, \sigma'_\kappa)\big) \\ \sigma'_\kappa = \Pi_\kappa\big(\varphi(\sigma_\delta, \sigma^t_\kappa)\big) \text{ avec } \sigma^t_\kappa = \Pi_\kappa\big([\![\Delta||\kappa]\!]^{\mathcal{H}}(\sigma_\kappa, \sigma_\delta)\big) \end{cases}
\end{aligned}
\tag{5.26}
$$

Rappelons que les fonctions Π_δ et Π_κ sont les projections des environnements hybrides vers les environnements discrets et continus, respectivement.

Remarque (1) La sémantique $[\![\Omega]\!]^{\mathcal{H}}$ est cohérente avec la sémantique dénotationnelle classique présentée au chapitre 2. En effet, si un programme P ne possède aucune instruction hybride, alors $[\![P||\kappa]\!]^{\mathcal{H}} = [\![P]\!]$.

Remarque (2) L'environnement continu est finalement calculé comme le point fixe de l'opérateur Γ_{F, y_0} dont la fonction F est construite au fur et à mesure du calcul de la sémantique du programme.

Remarque (3) En comparaison avec la sémantique discrète de la section 5.4, la sémantique hybride fait ressortir les calculs de point fixe sous-jacents à la fonction $[\![\boldsymbol{act}.k!c]\!]_\kappa$ (équation (5.15)) qui nécessite le calcul d'une sémantique continue et donc le calcul d'un point fixe. Nous avons choisi de rendre visible ce calcul de point fixe pour des raisons de cohérence avec la sémantique dénotationnelle standard.

5.6 Exemple de calcul de la sémantique

Nous illustrons le calcul de la sémantique hybride sur un système hybride à une variable continue dérivé du système des deux réservoirs. Ce système est composé d'un seul réservoir qui a deux tuyaux, un au fond du réservoir et un à une hauteur de H. Le tuyau du bas possède une vanne qui peut être ouverte ou fermée (voire figure 5.15). La hauteur d'eau dans le réservoir est alors donnée par les équations différentielles de la figure 5.15, à droite. La partie discrète du système hybride est un contrôleur qui ouvre et ferme la vanne du tuyau inférieur en fonction de la hauteur d'eau h. Nous ne nous intéresserons qu'aux deux premières secondes de l'exécution du contrôleur : on peut donc l'encoder en **H-SIMPLE** comme sur la figure 5.16. Nous supposerons que les capteurs fournissent une valeur par seconde.

Nous montrons maintenant le calcul de la sémantique de ce système hybride en détaillant les première itérations du calcul du point fixe. On commence donc le calcul de $[\![P||\kappa]\!]^{\mathcal{H}}$ en partant de

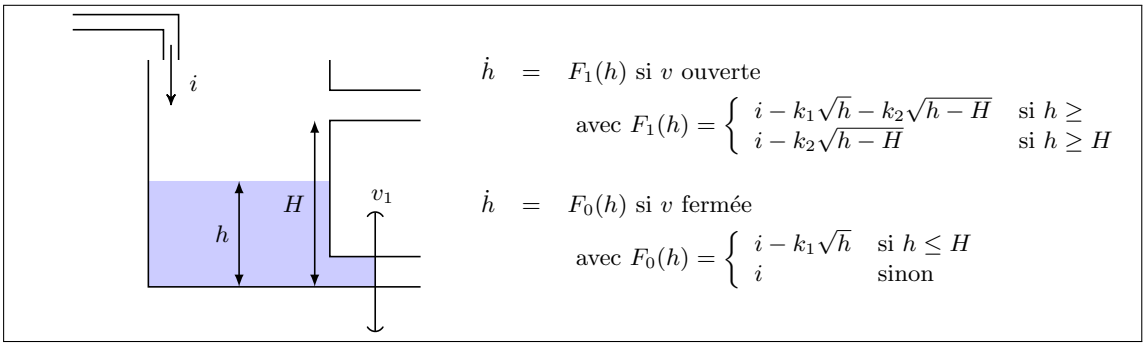


FIG. 5.15 – Système simplifié avec un seul réservoir.

```
 1  int main() {
 2    sensor x;          // capteur
 3    wait 1;
 4    sens.x?h;
 5    if (h>h_max)
 6      act.1!1;
 7    if (h<h1_min)
 8      act.1!0;
 9    wait (1);    // delai
10    sens.x?h;
11    if (h>h_max)
12      act.1!1;
13    if (h<h1_min)
14      act.1!0;
15    wait (1);    // delai
16    }
17  }
```

FIG. 5.16 – Contrôleur pour le système hybride à un réservoir.

l'état initial associant à chaque variable discrète $\perp$, à t la valeur 0, à act la valeur initiale $k_0 = 1$, à F la fonction F_{k_0} et à y le plus petit élément $\perp^0 \in \mathcal{D}^0$. L'état initial est donc $(\sigma_\kappa, \sigma_\delta)$ tel que $\forall X \in Var,\ \sigma_\delta(X) = \perp$, $\sigma_\delta.time = 0$, $\sigma_\delta.act = k_0$ et $\sigma_\kappa.F = F_1$, $\sigma_\kappa.y = \perp^0$.

Première itération Lors de l'exécution du programme, deux instructions ***sens*** sont exécutées, une à $t = 1$ et l'autre à $t = 2$. D'après la dénotation de cette instruction, la valeur de $\sigma_\kappa.y$ est donc, après la première exécution de ***sens***, changée en $\Gamma_{\sigma_\kappa.F, y_0}(\sigma_\kappa.y)$ qui est la fonction (a) sur la figure 5.17. Après la seconde instruction ***sens***, $\sigma_\kappa.y$ est la fonction (b) de la figure 5.17. On a donc $(\sigma'_\delta, \sigma'_\kappa) = [\![P||\kappa]\!]^{\mathcal{H}}(\sigma_\delta, \sigma_\kappa)$ avec :

- $\sigma'_\delta.h = 2.61348$;
- $\sigma'_\delta.t = 3$;
- $\sigma'_\kappa.F = F_0$;
- $\sigma'_\kappa.y$ donnée par la figure 5.17(b).

Comme $\sigma'_\kappa \neq \sigma_\kappa$, on recalcule $[\![P||\kappa]\!]^{\mathcal{H}}$ en partant cette fois-ci de l'environnement σ'_κ.

Deuxième itération L'état initial est donc maintenant $(\sigma_\delta, \sigma'_\kappa)$ où σ'_κ est calculée par la première itération, et σ_δ est l'état initial discret. L'environnement continu sera successivement changé en

les fonctions des figure 5.17(c) et 5.17(d). L'état final après la deuxième itération est donc $(\sigma'_\delta, \sigma''_\kappa)$ avec :

- $\sigma'_\delta.h = 2.84716$;
- $\sigma'_\delta.t = 3$;
- $\sigma''_\kappa.F = F_0$;
- $\sigma''_\kappa.y$ donnée par la figure 5.17(d).

On voit donc que l'état discret diffère de celui calculé après la première itération, la valeur de $\sigma_\delta.h$ se rapproche de la valeur définitive qui est $y_\infty(2)$, où y_∞ est la trajectoire réelle de la dynamique continue.

Troisième itération L'état initial est donc maintenant $(\sigma_\delta, \sigma''_\kappa)$ où σ''_κ est calculée par la seconde itération. L'environnement continu sera, après la première instruction ***sens***, la fonction de la figure 5.17(e) et après la seconde, $\sigma''_\kappa.y$ vaudra la fonction de la figure 5.17(f). L'état final est donc $(\sigma'_\delta, \sigma'''_\kappa)$ avec :

- $\sigma'_\delta.h = 2.85804$;
- $\sigma'_\delta.t = 3$;
- $\sigma'''_\kappa.F = F_0$;
- $\sigma'''_\kappa.y$ donnée par la figure 5.17(f).

On constate donc que la valeur de h s'approche de la vraie valeur. En deux itérations supplémentaires on obtient un point fixe, c'est-à-dire que la valeur de h ne change plus d'une itération à l'autre : la largeur de l'approximation continue est inférieure à la distance entre deux nombres flottants consécutifs. Si on suppose que la variable h_max vaut 2.85, alors on voit qu'au bout de 3 itérations seulement, la condition $h > h\_max$ est vérifiée et on voit donc que l'instruction $act.1!1$ de la ligne 12 est activée : notre sémantique calcule donc bien le comportement attendu du programme.

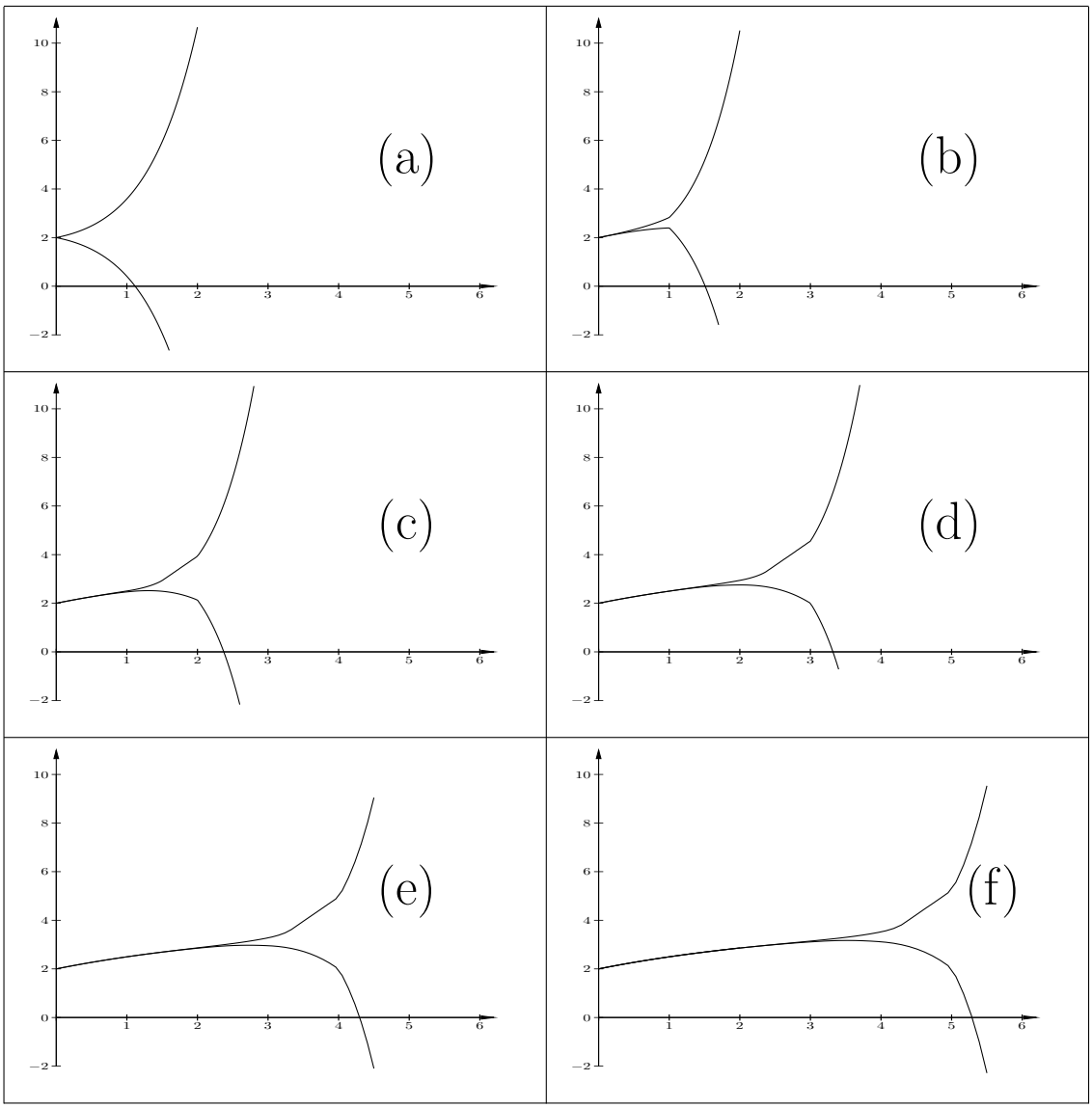


FIG. 5.17 – Évolution de l'environnement continu lors du calcul de la sémantique du système hybride à un réservoir.

Troisième partie

Abstrait

Dans cette partie, nous présentons des techniques permettant une analyse statique par interprétation abstraite des systèmes hybrides décrits précédemment. Dans un premier temps (chapitre 6), nous montrerons que les algorithmes d'intégration garantie peuvent être vus comme des fonctions d'abstraction au sens de l'interprétation abstraite. Nous introduirons notamment le concept d'équation différentielle intervalle qui généralise les équations différentielles en prenant en compte des incertitudes sur les conditions initiales. Nous décrirons également une nouvelle méthode d'intégration garantie, nommée GRKLib, qui obtient des encadrements garantis de la solution d'une équation différentielle en se basant sur un algorithme d'intégration numérique. Dans un deuxième temps (chapitre 7), nous expliquerons comment on peut calculer une abstraction du système hybride dans son ensemble en séparant l'analyse de la partie continue et l'analyse de la partie discrète.

CHAPITRE 6

Abstraction de la partie continue

Dans ce chapitre, nous présentons une abstraction de la partie continue des systèmes hybrides décrits au chapitre 5. Notre but est d'utiliser les algorithmes d'intégration garantie présentés au chapitre 3 comme une abstraction, au sens de l'interprétation abstraite, de la sémantique de la partie continue. Nous commençons donc par définir intuitivement les conditions que doit remplir une abstraction de l'environnement continu pour rentrer dans le cadre de l'interprétation abstraite (section 6.1), puis nous définissons formellement le domaine abstrait des fonctions en escalier à valeurs dans les intervalles (section 6.2) et nous donnons une représentation efficace de celles-ci sous forme d'une conjonction de contraintes. Les opérateurs abstraits d'union et d'intersection seront définis sur ce domaine des contraintes. Enfin, nous montrons que les algorithmes décrits au chapitre 3 forment des abstractions valides (section 6.3). Par ailleurs, nous expliquerons en détail un nouvel algorithme d'intégration garantie, nommé GRKLib, qui vise à limiter le wrapping effect inhérent aux méthodes à base d'arithmétique d'intervalles (sections 6.4 à 6.5).

6.1 But de l'abstraction de la partie continue

La partie continue de notre modèle de systèmes hybrides est décrite par un ensemble d'équations différentielles. La sémantique continue est alors une fonction continue qui oscille entre les solutions de ces équations différentielles selon les instants auxquels le programme agit sur les actionneurs. Dans ce chapitre, nous ne nous intéressons qu'au cas où l'environnement continu est construit à partir d'*une* équation différentielle contenant éventuellement des paramètres discrets constants dans le temps qui permettent d'introduire éventuellement des incertitudes à l'équation différentielle. Nous verrons au chapitre suivant comment construire une abstraction complète de l'environnement continu à partir d'une abstraction du programme et d'une abstraction de chaque équation différentielle. Nous nous intéressons donc à une équation différentielle autonome, de degré 1 et de dimension n quelconque, que nous noterons $\dot{y} = F(y, \vec{k})$ où $\vec{k} \in \mathbb{R}^m$ est un ensemble de paramètres constants dans le temps. Nous noterons $P := \big(\dot{y} = F(y, \vec{k}),\ y(0) = y_0\big)$ le problème de Cauchy sous-jacent, avec $y_0 \in \mathbb{R}^n$. La sémantique de P est clairement la solution de ce problème ; la question de l'existence de la solution n'est pas discutée ici, on pourra se référer au chapitre 5 pour une discussion détaillée sur ce problème. Nous supposerons en particulier que F est globalement lipschitzienne pour toutes les valeurs possibles des paramètres $\vec{k}$. La sémantique $[\![P]\!]$ de P est donc définie par :

$$[\![P]\!] : \left\{ \begin{array}{ccc} \mathbb{R}_+ & \rightarrow & \mathbb{R}^n \\ t & \mapsto & y_\infty(t) \end{array} \right. \quad \text{tel que } \forall t \in \mathbb{R}_+,\ \dot{y}_\infty(t) = F\big(y_\infty(t), \vec{k}\big) \text{ et } y_\infty(0) = y_0\ . \tag{6.1}$$

6.1.1 Sémantique collectrice d'une équation différentielle intervalle

Comme nous l'avons fait pour la sémantique des programmes impératifs, nous définissons maintenant une sémantique collectrice pour les équations différentielles. Cette sémantique définit l'ensemble des comportements possibles décrits par une seule équation différentielle, tout comme la sémantique dénotationnelle collectrice définit l'ensemble des comportements possibles d'un programme pour un ensemble de valeurs d'entrée. Cependant, un problème de Cauchy définit, nous l'avons vu, un seul comportement qui est sa solution maximale. Nous étendons donc le concept de problème de Cauchy en autorisant les conditions initiales à être données sous forme d'intervalle (ce qui définit un ensemble de valeurs initiales possibles) et l'équation différentielle à avoir des paramètres fournis sous forme d'intervalle également (ce qui définit un ensemble de comportements possibles). Nous introduisons donc le concept d'*équation différentielle intervalle*.

Définition 6.1 (Équation différentielle intervalle) Une équation différentielle intervalle (EDI) de dimension n est une relation entre une fonction $y : \mathcal{D} \to \mathbb{R}^n$ définie sur un ouvert $\mathcal{D}$ et sa dérivée temporelle $\dot{y} : \mathcal{D} \to \mathbb{R}^n$, donnée par :

$$\dot{y}(t) = F(y(t), [\vec{k}]) \tag{6.2}$$

où $[\vec{k}] \in I^m_{\mathbb{R}}$ est un vecteur d'intervalles de dimension m et $F : \mathbb{R}^{n+m} \to \mathbb{R}^n$. Par convention, nous noterons souvent l'EDI $\dot{y} = F(y, [\vec{k}])$.

Exemple 6.1 L'équation $\dot{y} = [-2, -1] \times y$ définit une EDI de dimension 1.

Dans une EDI, nous autorisons donc les paramètres de la fonction F à être des intervalles, ce qui nous permet d'obtenir des comportements différents. Nous étendons naturellement cette définition aux *Problèmes de Cauchy intervalle*.

Définition 6.2 (Problème de Cauchy intervalle) Étant donnée une EDI $\dot{y} = F(y, [\vec{k}])$ et une condition initiale donnée sous forme d'intervalle $y(0) \in [y_0]$, le problème de Cauchy intervalle (PCI) consiste à trouver un ouvert $D \subseteq \mathbb{R}$ qui contienne 0 et l'ensemble des solutions aux problèmes de Cauchy classiques donnés par $\dot{y} = F(y, \vec{k})$, $y(0) = y_0$ pour tout $\vec{k} \in [\vec{k}]$ et $y_0 \in [y_0]$. Nous noterons $\mathcal{PCI}$ l'ensemble de tous les problèmes de Cauchy intervalle.

La sémantique collectrice d'un PCI $P \in \mathcal{PCI}$, $P := \big(\dot{y} = F(y, [\vec{k}],\ y(0) \in [y_0]\ \big)$, est donc un ensemble de fonctions continues donné par :

$$(\!|P|\!) = \Big\{ y \in \mathcal{C}^0\big(\mathbb{R}_+ \to \mathbb{R}^n\big)\ :\ \exists k \in [\vec{k}], y_0 \in [y_0]\,,\ y = [\![\dot{y} = F(y,k),\ y(0) = y_0]\!] \Big\} \tag{6.3}$$

Exemple 6.2 Le PCI $P := \big(\dot{y} = [-2, -1] \times y,\ y(0) \in [0.5, 3]\ \big)$ définit l'ensemble des fonctions de la forme $y(t) = y_0.e^{-k.t}$ avec $y_0 \in [0.5, 3]$ et $k \in [-2, -1]$. La figure 6.1 représente graphiquement cet ensemble.

6.1.2 Caractéristiques nécessaires pour le domaine abstrait

On voit donc que la sémantique collectrice d'un problème de Cauchy intervalle est un ensemble, potentiellement infini, de fonctions continues. Cet ensemble n'est généralement ni calculable ni représentable en machine. Il n'est pas calculable car, comme nous l'avons vu au chapitre 5, la sémantique concrète d'une EDO nécessite le calcul d'un point fixe via une itération infinie de l'opérateur de Picard modifié qui n'est lui-même pas calculable exactement en machine. Nous verrons que les algorithmes d'intégration garantie fournissent une abstraction calculable de ce

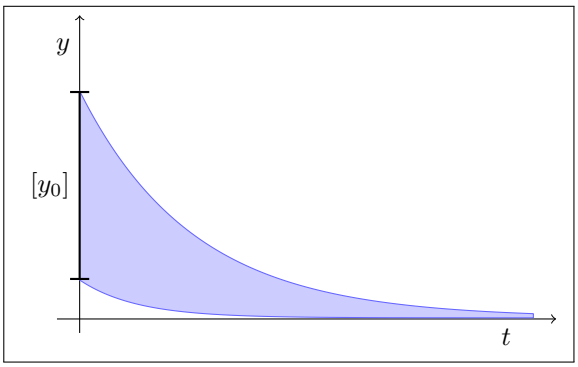


FIG. 6.1 – Solutions du PCI de l'exemple 6.2.

point fixe. Les éléments du domaine concret ne sont pas non plus représentables en machine, car un ensemble de fonctions continues associe, à chaque instant $t \in \mathbb{R}_+$, un ensemble potentiellement infini de nombres réels. Le domaine abstrait doit donc effectuer une partition (ou un recouvrement, si une partition n'est pas possible) de l'ensemble des instants $t \in \mathbb{R}_+$ et, pour chaque élément de cette partition, surapproximer par un objet représentable en machine l'ensemble des valeurs prises par chacune des fonctions. Nous définissons dans la suite le domaine des fonctions en escalier à valeurs dans le domaine des hyperrectangles, qui vérifie ces deux conditions, puis nous définissons le domaine des contraintes qui permet une représentation efficace des fonctions en escalier.

6.2 Domaines abstraits

Dans la suite, nous noterons $\mathcal{PC}^0 = \mathcal{P}\left(\mathcal{C}^0\left(\mathbb{R}_+ \to \mathbb{R}^n\right)\right)$ le domaine concret des ensembles de fonctions continues de $\mathbb{R}_+$ vers $\mathbb{R}^n$. Clairement, $\mathcal{PC}^0$ possède une structure de treillis complet avec les opérateurs ensemblistes classiques $\subseteq$, $\cup$ et $\cap$.

Définition 6.3 (Treillis des intervalles et treillis des hyperrectangles) Le treillis complet des intervalles fermés, non bornés de $\mathbb{R}$ est noté $\mathbb{IR}$:

$$\mathbb{IR} = \left\{ [a,b] \ : \ a,b \in \mathbb{R},\ a \leq b \right\} \cup \left\{ [a,\infty[\ : \ a \in \mathbb{R} \right\} \cup \left\{]-\infty,a] \ : \ a \in \mathbb{R} \right\} \cup \{]-\infty,\infty[\} \tag{6.4}$$

L'élément maximal $]-\infty,\infty[$ sera noté $\top_{\mathbb{IR}}$, et nous ajouterons à $\mathbb{IR}$ un élément minimal $\bot_{\mathbb{IR}} = \emptyset$. Nous utiliserons l'ordre d'inclusion $\sqsubseteq$ ainsi que l'intersection $\sqcap$ et l'union $\sqcup$ usuelle sur ces intervalles.
Nous noterons $\mathbb{IR}^n$ le treillis complet des hyperrectangles de dimension n, c'est-à-dire l'ensemble des produits cartésiens de n intervalles *non vides*. Le plus petit élément de $\mathbb{IR}^n$ sera noté $\bot_{\mathbb{IR}^n}$. La relation d'ordre $\sqsubseteq^n$ et l'union $\sqcup^n$ sont des extensions coordonnée-par-coordonnée de l'ordre et de l'union sur $\mathbb{IR}$. L'intersection $\sqcap^n$ est définie de telle sorte que, pour $x, y \in \mathbb{IR}^n$, si l'intersection des intervalles de la i-ième coordonnée (pour $0 \leq i \leq n$) de x et de y est vide, alors $x \sqcap^n y = \bot_{\mathbb{IR}^n}$.

Remarque Le treillis $\mathbb{IR}^n$ est le produit cartésien classique de $\mathbb{IR}$ avec lui-même n fois, sauf que nous identifions en $\bot_{\mathbb{IR}^n}$ tous les vecteurs ayant une coordonnée égale à l'ensemble vide $\bot_{\mathbb{IR}}$. Par exemple, dans $\mathbb{IR}^2$, $\bot_{\mathbb{IR}^2}$ représente tous les vecteurs de la forme $([x], \emptyset)$ ou $(\emptyset, [x])$ avec $[x] \in \mathbb{IR}$. On a alors notamment $\left([1,2], [1,2] \right) \sqcap^2 \left([0,1.5], [3,4] \right) = \bot_{\mathbb{IR}^2}$.

L'élément maximal de $\mathbb{IR}^n$ est l'hyperrectangle dont toutes les coordonnées sont égales au vecteur $]-\infty,\infty[$, nous le noterons $\top_{\mathbb{IR}^n}$. Pour tout $x \in \mathbb{IR}^n$, nous notons $\underline{x} \in \mathbb{R}^n$ (respectivement $\overline{x} \in \mathbb{R}^n$) le vecteur de nombres réels obtenu en prenant pour chaque coordonnée de x la borne inférieure (respectivement supérieure) de l'intervalle associé.

6.2.1 Première abstraction : fonctions en escalier à valeurs dans $\mathbb{IR}^n$

Comme nous l'avons dit, nous utiliserons des fonctions en escalier à valeur dans $\mathbb{IR}^n$ pour abstraire un ensemble de fonctions continues $P \in \mathcal{PC}^0$.

Définition 6.4 (Domaine abstrait $\mathcal{FS}^n$) Le domaine abstrait est le domaine $\mathcal{FS}^n$ des fonctions en escalier à support fini et à valeur dans $\mathbb{IR}^n$:

$$\mathcal{FS}^n = \left\{ y : \mathbb{R}_+ \to \mathbb{IR}^n \ : \begin{array}{c} \exists N \in \mathbb{N},\ \exists t_0 = 0 < t_1 < \cdots < t_{N-1} < t_N \text{ tels que} \\ -\ \forall i \in [0, N-1],\ y \text{ est constante sur } [t_i, t_{i+1}[\\ -\ y \text{ est constante sur } [t_N, \infty[\end{array} \right\} .$$

On munit aisément $\mathcal{FS}^n$ d'une structure de treillis. L'ordre partiel $\sqsubseteq$ est une extensions point-à-point de l'ordre $\sqsubseteq^n$ sur $\mathbb{IR}^n$: $\forall f, g \in \mathcal{FS}^n,\ f \sqsubseteq g \Leftrightarrow \forall t \in \mathbb{R}_+,\ f(t) \sqsubseteq^n g(t)$. On ajoute alors à $\mathcal{FS}^n$ un élément minimal noté $\bot_{fsn}$ vérifiant $\forall f \in \mathcal{FS}^n,\ \bot_{fsn} \sqsubseteq f$. L'élément maximal de $\mathcal{FS}^n$ est clairement la fonction constante égale à $\top_{\mathbb{IR}^n}$, nous le noterons $\top_{fsn}$.
On définit également l'opérateur d'union $\sqcup$ entre deux fonctions en escalier $f, g \in \mathcal{FS}^n$ par une extension point-à-point de l'opérateur d'union défini sur $\mathbb{IR}^n$. L'intersection $\sqcap$ de $f, g \in \mathcal{FS}^n$ est une extension point-à-point de $\sqcap^n$ sauf que s'il existe $t_0 \in \mathbb{R}_+$ tel que $f(t_0) \sqcap^n g(t_0) = \bot_{\mathbb{IR}^n}$, alors $f \sqcap g = \bot_{fsn}$.

Remarque (1) Le treillis $\mathcal{FS}^n$ est un extension classique du treillis $\mathbb{IR}^n$, sauf que nous identifions en $\bot_{fsn}$ toutes les fonctions prenant la valeur $\bot_{\mathbb{IR}^n}$ pour un $t_0 \in \mathbb{R}_+$. Cela explique que nous devons définir l'intersection de façon non standard.

Remarque (2) Le treillis $\mathcal{FS}^n$ n'est pas un treillis complet. En effet, la famille de fonctions en escalier $(f_n)_{n\in\mathbb{N}}$ définies par :

$$f_n(t) = \left\{ \begin{array}{rcl} [1,1] & \text{si} & t \geq 1 \\ \left[0, \frac{\lfloor n \times t \rfloor + 1}{n}\right] & \text{si} & 0 \leq t < 1 \end{array} \right. \tag{6.5}$$

ne possède pas de plus grand minorant : pour tout $t \in [0,1]$, on a $\lim_{n\to\infty} f_n(t) = [0,t]$, de telle sorte que si un plus grand minorant f existait, il devrait vérifier $\forall t \in [0,1]\,,\ f(t) = [0,t]$, f ne serait donc pas une fonction en escalier finie. Une représentation graphique de la famille de fonctions $(f_n)_{n\in\mathbb{N}}$ est donnée à la figure 6.2. Le treillis $\mathcal{FS}^n$ n'est pas non plus un CPO.

Concrétisation des fonctions en escalier Nous établissons maintenant le lien entre le treillis concret $\mathcal{PC}^0$ et le treillis abstrait $\mathcal{FS}^n$ via une fonction de concrétisation γ. Intuitivement, une fonction en escalier à valeurs dans les intervalles représente l'ensemble des fonctions continues qui restent dans les bornes définies par la fonction en escalier. Cette intuition est formalisée par la définition 6.5.

Définition 6.5 (Fonction de concrétisation) La fonction de concrétisation γ entre les fonctions en escalier $\mathcal{FS}^n$ et les ensembles de fonctions continues $\mathcal{PC}^0$ est définit par :

$$\gamma : \left\{ \begin{array}{rcl} \mathcal{FS}^n & \to & \mathcal{PC}^0 \\ f & \mapsto & \left\{ y \in \mathcal{C}^0(\mathbb{R}_+ \to \mathbb{R}^n) \ :\ \forall t \in \mathbb{R}_+,\ y(t) \in f(t) \right\} \end{array} \right. \tag{6.6}$$

On définit de plus $\gamma(\bot_{fsn}) = \emptyset$. Remarquons que, si une fonction $f \in \mathcal{FS}^n$ a deux marches successives disjointes (comme par exemple sur la figure 6.3), alors $\gamma(f) = \emptyset$:

$$\left(\exists t_0 \in \mathbb{R}_+,\ \Big(\lim_{\substack{t \to t_0 \\ t < t_0}} f(t) \Big) \cap \Big(\lim_{\substack{t \to t_0 \\ t > t_0}} f(t) \Big) = \emptyset \right) \Longrightarrow \gamma(f) = \emptyset\,.$$

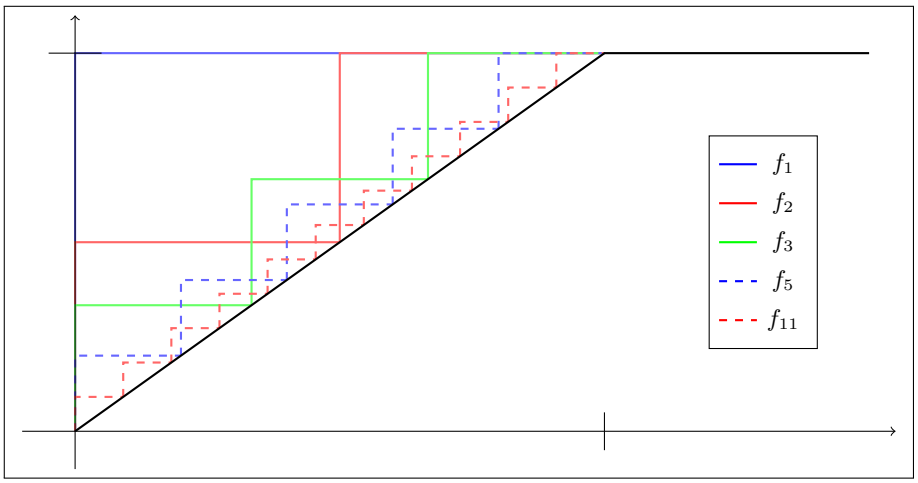


FIG. 6.2 – Représentation de la famille de fonctions définie par l'équation (6.5).

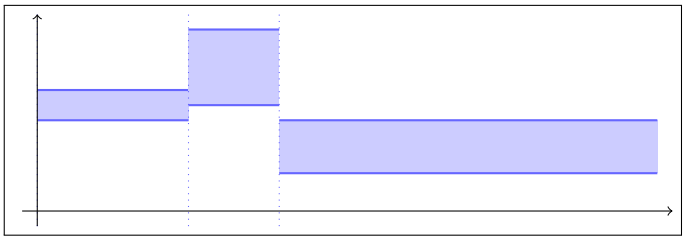


FIG. 6.3 – Exemple de fonction en escalier f telle que $\gamma(f) = \emptyset$;

Un exemple de fonction abstraite (la zone rouge clair délimitée par des marches foncées) est donné sur la figure 6.4. Cette fonction abstraite définit un ensemble de fonctions concrètes, dont la fonction en traits pointillés noirs. Notons que la sémantique du PCI de l'exemple 6.2 (région bleue sur la figure) est incluse dans cette abstraction (région rose).

Fonction d'abstraction et critère de sûreté. Clairement, il n'existe pas de meilleure abstraction $\alpha : \mathcal{PC}^0 \to \mathcal{FS}^n$, c'est-à-dire qu'on ne peut pas trouver de fonction α telle que $(\mathcal{PC}^0, \subseteq) \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} (\mathcal{FS}^n, \sqsubseteq)$ soit une correspondance de Galois (le problème est le même que pour le domaine des polyèdres [CH78]). Cependant, il n'est pas nécessaire d'avoir une correspondance de Galois pour définir une interprétation abstraite correcte [CC92a]. Il suffit en effet que la fonction de concrétisation γ et la sémantique abstraite $(\!|.|\!)^\sharp$ vérifient que pour tout PCI P, $(\!|P|\!) \subseteq \gamma((\!|P|\!)^\sharp)$. Nous donnons ci-dessous un critère simple à vérifier pour qu'une sémantique abstraite soit considérée comme sûre.

Définition 6.6 (Critère de sûreté pour la sémantique abstraite) Une sémantique abstraite $(\!|.|\!)^\sharp$ vérifie le critère de validité CS1 si et seulement si, pour tout PCI P, elle vérifie l'équation (6.7), avec $f = (\!|P|\!)^\sharp$:

$$\forall t \in \mathbb{R}_+,\ \underline{f(t)} \leq inf\{y(t)\ :\ y \in (\!|P|\!)\} \quad \text{et} \quad sup\{y(t)\ :\ y \in (\!|P|\!)\} \leq \overline{f(t)}\ . \tag{6.7}$$

Remarque Dans la définition 6.6, $\underline{f(t)}$ et $\overline{f(t)}$ sont des vecteurs de nombres réels, de même que $y(t)$ pour $y \in (\!|P|\!)$. L'ordre sur les vecteurs $\leq$ s'entend alors coordonnée-par-coordonnée.

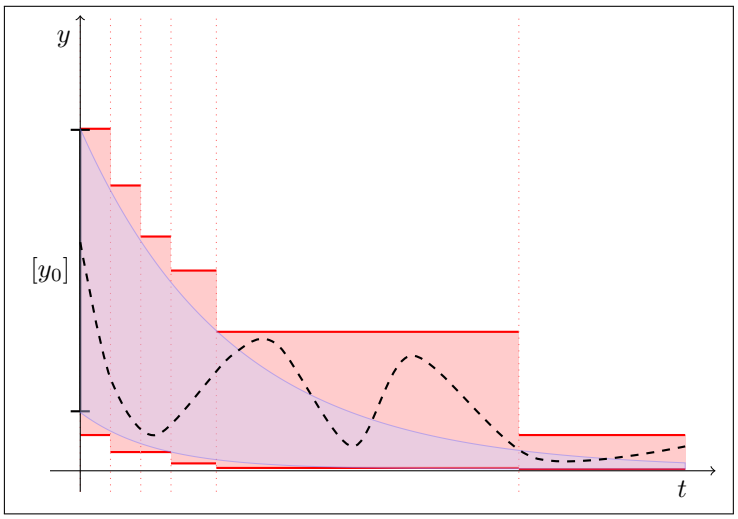


FIG. 6.4 – Fonctions en escalier abstraites.

Théorème 6.1 *Si* $(\!|.|\!)^\sharp$ *vérifie le critère de la formule* (6.7)*, alors pour tout problème de Cauchy intervalle* P*, on a* $(\!|P|\!) \subseteq \gamma\big((\!|P|\!)^\sharp\big)$.

Preuve Supposons que $(\!|.|\!)^\sharp$ vérifie l'équation (6.7). Soit $P \in \mathcal{PCI}$ un problème de Cauchy intervalle et $P' = \gamma\big((\!|P|\!)^\sharp\big)$. Nous voulons montrer que $P \subseteq P'$. Soit $y \in P$, on sait d'après l'équation (6.7) que $\forall t \in \mathbb{R}_+,\ y(t) \in (\!|P|\!)^\sharp(t)$. D'après la définition de la fonction de concrétisation (équation (6.6)), on a donc $y \in \gamma\big((\!|P|\!)^\sharp\big)$, d'où $P \subseteq P'$. □

6.2.2 Deuxième abstraction : représentation de $\mathcal{FS}^n$ par des contraintes

Une fonction abstraite $f \in \mathcal{FS}^n$ représente un ensemble infini de fonctions continues $\mathcal{Y} \in \mathcal{PC}^0$. Cependant, nous n'avons pas encore défini de représentation efficace des fonctions en escalier ; en particulier, les opérateurs d'union et d'intersection que nous avons défini ne sont pas calculables en l'état car ils nécessitent de calculer une infinité d'unions (ou d'intersections) d'intervalle. De plus, la comparaison entre deux fonctions en escalier n'est pas simples : deux fonctions peuvent être égales mais avoir des marches de longueurs différentes ; il est alors difficile de les comparer. Nous présentons donc maintenant une abstraction des fonctions en escalier sous forme d'une conjonction de contraintes qui permet une représentation et un calcul efficace sur les éléments abstraits. Cette abstraction (qui est en fait une bijection de Galois) consiste à ne conserver que les instants où la fonction change de valeur et permet une représentation unique des éléments de $\mathcal{FS}^n$.

Représentation canonique des fonctions de $\mathcal{FS}^n$ Nous introduisons un nouveau type de contraintes, appelées contraintes de changements d'état, qui sont une extension des contraintes définies par Bertrane [Ber05].

Définition 6.7 (Domaine des contraintes de changement d'état) Un *prédicat de changement d'état* de dimension n est un prédicat de la forme $t : [x]$ avec $t \in \mathbb{R}_+$ et $[x] \in \mathbb{IR}^n$. Une *contrainte de changement d'état* est une conjonction positive de prédicats de changement d'état. Nous noterons $\mathbb{FS}^n$ l'ensemble de ces contraintes.

$$\mathbb{FS}^n = \left\{ \bigwedge_{0 \le i \le N} t_i : [x_i] \text{ tel que } N \in \mathbb{N},\ \forall i \in [0, N],\ t_i \in \mathbb{R}_+,\ [x_i] \in \mathbb{IR}^n \right\} \cup \{\mathtt{ff}^\sharp\} \tag{6.8}$$

$\mathtt{ff}^\sharp$ est l'élément minimal de $\mathbb{FS}^n$. Nous noterons $\mathtt{tt}^\sharp = 0 :]-\infty, \infty[$ l'élément maximal.

Intuitivement, une contrainte de changement d'état représente toutes les discontinuités d'une fonction en escalier. Ainsi, la contrainte $0 : [0,1] \wedge 1 : [1,1]$ représente la fonction en escalier valant $[0,1]$ pour tout $t \in [0,1[$ et 1 pour tout $t \geq 1$ (il s'agit de la fonction f_1 de la figure 6.2). Cette représentation n'est pas unique, ce qui est problématique pour l'utilisation efficace de ce domaine. En effet, on voit bien que les contraintes $0 : [0,1] \wedge 1 : [1,2]$ et $0 : [0,1] \wedge 1 : [1,2] \wedge 2 : [1,2]$ représentent la même fonction $f = \lambda t.if\ (t < 1)\ then\ [0,1]\ else\ [1,2]$. Dans le deuxième cas, la dernière contrainte n'apporte aucun information supplémentaire, on peut l'éliminer ; nous choisirons donc la première contrainte comme représentant canonique de la fonction f. Par ailleurs, les contraintes $0 : [0,1] \wedge 1 : [1,2]$ et $1 : [1,2] \wedge 0 : [0,1]$ représentent toutes les deux la fonction f. Pour des questions de performance de la comparaison, nous choisissons de garder la première comme représentant ; nous décidons donc de toujours garder les contraintes triées par temps croissant. Enfin, par soucis de cohérence, nous n'autorisons pas une conjonction de deux contraintes avec le même temps $t : [x] \wedge t : [y]$; dans ce cas, nous ne garderons que le premier prédicat. Ces trois règles permettent de définir une fonction de normalisation *Norm* sur $\mathbb{FS}^n$.

Définition 6.8 (Normalisation des fonctions de $\mathbb{FS}^n$) La fonction *Norm* de normalisation des fonctions de $\mathbb{FS}^n$ est donnée par l'algorithme suivant :

Entrée : $\phi = \bigwedge_{i \in [0,N]} t_i : [x_i] \in \mathbb{FS}^n$
Résultat : ϕ sous forme normalisée
début
  Si $\phi = \mathtt{tt}^\sharp$ *ou* $\phi = \mathtt{ff}^\sharp$ **renvoyer** ϕ;
  Tri les contraintes avec l'ordre $t_i : [x_i] \leq t_j : [x_j] \Leftrightarrow (t_i \leq t_j) \vee (i \leq j)$;
  $\psi = t_0 : [x_0]$;
  $t = t_0;\ [x] = [x_0]$;
  pour $1 \leq i \leq N-1$ **faire**
    si $t \neq t_i$ *et* $[x] \neq [x_i]$ **alors**
      $\psi = \psi \wedge t_i : [x_i]$;
      $t = t_i;\ [x] = [x_i]$;
    finsi
  finpour
  renvoyer ψ ;
fin

Nous noterons $\mathbb{FS}^n_{\mathrm{n}}$ l'ensemble des contraintes normalisées, $\mathbb{FS}^n_{\mathrm{n}} = \{f \in \mathbb{FS}^n : Norm(f) = f\}$. Entre deux contraintes normalisées $f = \bigwedge_{0 \leq i \leq N} t_i : [x_i] \in \mathbb{FS}^n_{\mathrm{n}}$ et $g = \bigwedge_{0 \leq i \leq M} u_i : [y_i] \in \mathbb{FS}^n_{\mathrm{n}}$, on peut définir une égalité structurelle $=^s$ par :

$$f =^s g \Leftrightarrow \begin{cases} f = g = \mathtt{tt}^\sharp \ ou \\ f = g = \mathtt{ff}^\sharp \ ou \\ N = M \wedge \forall i \in [0,N]\,,\ t_i = u_i \wedge [x_i] = [y_i] \end{cases} .$$

Remarque La fonction de normalisation sert essentiellement à réorganiser les prédicats de changement d'état de telle sorte qu'ils soient triés par temps croissants. De plus, elle élimine les prédicats superflus. Notons que cette fonction n'introduit aucun nouveau prédicat, comme le montre la proposition 6.2

Proposition 6.2 *Soit $\phi \in \mathbb{FS}^n$, $\phi = \bigwedge_{0 \leq i \leq N} t_i : [x_i]$. Alors, il existe $M \in [0,N]$ et une fonction $\sigma : [0,M] \to [0,N]$ telle que*

$$Norm(\phi) = \bigwedge_{0 \leq i \leq M} t_{\sigma(i)} : \left[x_{\sigma(i)}\right] .$$

Preuve La preuve de cette proposition est simple mais fastidieuse : on doit faire une analyse détaillée de l'algorithme de la définition 6.8 et on prouve que la forme normale décrite par cette proposition est un invariant de l'algorithme.

□

Proposition 6.3 (Relation d'équivalence) *La fonction de normalisation Norm induit une relation d'équivalence sur* $\mathbb{FS}^n$ *définie par* $f \equiv g \Leftrightarrow Norm(f) =^s Norm(g)$.

Preuve D'après la définition 6.8, l'égalité structurelle $=^s$ est une relation d'équivalence (elle est clairement réflexive, transitive et symétrique). On en conclut immédiatement que $\equiv$ est également une relation d'équivalence. □

Nous travaillerons désormais dans l'ensemble réduit $\mathbb{FS}^n_{/\equiv}$ et nous identifierons une contrainte $f \in \mathbb{FS}^n$ avec sa forme normalisée $Norm(f) \in \mathbb{FS}^n_{\mathrm{n}}$. Par abus de notation, nous noterons toujours $\mathbb{FS}^n$ pour $\mathbb{FS}^n_{/\equiv}$. Nous définissons maintenant un ordre partiel sur les contraintes normalisées.

Remarque Par un léger abus de notation, pour toute contrainte $f = \bigwedge_{0\leq i\leq N} t_i : [x_i] \in \mathbb{FS}^n$, nous noterons $t_{N+1} = +\infty$.

Définition 6.9 (Ordre partiel sur $\mathbb{FS}^n$) Soit $f = \bigwedge_{0\leq i\leq N} t_i : [x_i] \in \mathbb{FS}^n$ et $g = \bigwedge_{0\leq i\leq M} u_i : [y_i] \in \mathbb{FS}^n$. On définit la relation d'ordre partiel $f \Rightarrow^\sharp g$ par :

$$f \Rightarrow^\sharp g \Longleftrightarrow \forall (i,j) \in [0,N]\times[0,M]\,,\ [t_i,t_{i+1}[\cap[u_j,u_{j+1}[\neq\emptyset \Rightarrow [x_i] \sqsubseteq^n [y_j]\ . \tag{6.9}$$

On aura de plus $\forall \phi \in \mathbb{FS}^n$, $\mathtt{ff}^\sharp \Rightarrow \phi$.

Exemple 6.3 La figure 6.5 représente deux cas de contraintes comparables et incomparables. En dimension 1, une contrainte $\bigwedge_{0\leq i\leq n} t_i : [x_i]$ est représentée graphiquement par une suite de rectangles séparés par une ligne verticale pointillée représentant l'instant de changement d'état. Sur la figure de gauche, la contrainte rouge est plus petite que la contrainte bleue.

Proposition 6.4 $\Rightarrow^\sharp$ *est un ordre partiel sur* $\mathbb{FS}^n$.

Preuve Dans toute cette preuve, $\phi, \psi \in \mathbb{FS}^n$ seront deux contraintes avec $\phi \neq \mathtt{ff}^\sharp$ et $\psi \neq \mathtt{ff}^\sharp$. Nous poserons $\phi = \bigwedge_{0\leq i\leq N} t_i : [x_i]$ et $\psi = \bigwedge_{0\leq i\leq M} u_i : [y_i]$.
Nous devons montrer que la relation $\Rightarrow^\sharp$ vérifie les conditions de la définition 2.1. Nous montrons donc qu'elle est :

(i) réflexive, c'est-à-dire que $\phi \Rightarrow^\sharp \phi$. Clairement, pour tout $i,j \in [0,N]$, $[t_i,t_{i+1}[\cap[t_j,t_{j+1}[\neq\emptyset \Leftrightarrow i=j$, donc on a bien $[x_i] \sqsubseteq^n [x_j]$, et donc $\phi \Rightarrow^\sharp \phi$.

(ii) transitive. On suppose que $\phi \Rightarrow^\sharp \psi$, et soit $\zeta \in \mathbb{FS}^n$ telle que $\psi \Rightarrow^\sharp \zeta$. On veut montrer que $\phi \Rightarrow^\sharp \zeta$. On pose $\zeta = \bigwedge_{0\leq i\leq P} v_i : [z_i]$. Soit $(i,k) \in [0,N]\times[0,P]$ tels que $[t_i,t_{i+1}]\cap[v_k,v_{k+1}] \neq \emptyset$, il existe donc $t \in \mathbb{R}_+$ tel que $t \in [t_i,t_{i+1}]\cap[v_k,v_{k+1}]$. Soit $j \in [0,N]$ tel que $t \in [u_j,u_{j+1}]$. Alors, $[t_i,t_{i+1}]\cap[u_j,u_{j+1}] \neq \emptyset$ et $[u_j,u_{j+1}]\cap[v_k,v_{k+1}] \neq \emptyset$, donc $[x_i] \sqsubseteq^n [y_j] \sqsubseteq^n [z_k]$, d'où $[x_i] \sqsubseteq^n [z_k]$, et donc $\phi \Rightarrow^\sharp \zeta$.

(iii) antisymétrique : on suppose que $\phi \Rightarrow^\sharp \psi$ et $\psi \Rightarrow^\sharp \phi$. On peut toujours supposer que $N \geq M$. On montre alors que :

1. $\forall i \leq M,\ t_i = u_i \wedge [x_i] = [y_i]$. Supposons que $\exists i \in [0,M]$ tel que $t_i \neq u_i$, et posons $i_0 = min\{i \in [0,M]\ :\ t_i \neq u_i\}$, on sait que $i_0 > 0$. On peut toujours supposer que $t_{i_0} < u_{i_0}$. On est alors dans la situation suivante : $u_{i_0-1} = t_{i_0-1} < t_{i_0} < u_{i_0}$. On a alors :

$$[u_{i_0-1},u_{i_0}[\cap[t_{i_0-1},t_{i_0}[\neq\emptyset \Rightarrow y_{i_0-1} \sqsubseteq^n x_{i_0-1} \wedge x_{i_0-1} \sqsubseteq^n y_{i_0-1} \Rightarrow y_{i_0-1} = x_{i_0-1}$$

$$[u_{i_0-1},u_{i_0}[\cap[t_{i_0},t_{i_0+1}[\neq\emptyset \Rightarrow y_{i_0} \sqsubseteq^n x_{i_0-1} \wedge x_{i_0-1} \sqsubseteq^n y_{i_0} \Rightarrow y_{i_0} = x_{i_0-1}$$

On a alors $[y_{i_0}] = [y_{i_0-1}]$ ce qui est impossible car ϕ est normalisée. On a donc $\forall i \in [0,M]\,,\ t_i = u_i$ et par antisymétrie de la relation $\sqsubseteq^n$, on en déduit que $[x_i] = [y_i]$.

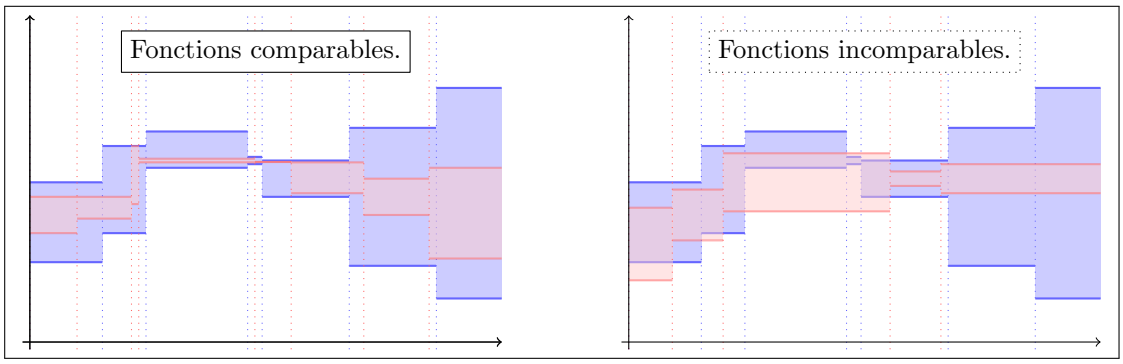


FIG. 6.5 – Ordre partiel dans le domaine des fonctions en escalier à valeur intervalle.

2. $N = M$. Supposons que $N > M$, on sait que $t_M = u_M$ et $[y_M] = [x_M]$. On a alors $[t_{M+1}, t_{M+2}[\cap[x_M, \infty[\neq \emptyset$ (si $N = M+1$, on pose $t_{M+2} = \infty$), et donc $[x_{M+1}] \sqsubseteq^n [y_M]$ et $[y_M] \sqsubseteq^n [x_{M+1}]$, d'où $[x_{M+1}] = [y_M] = [x_M]$, on a donc $[x_{M+1}] = [x_M]$, ce qui est impossible car ϕ est normalisée.

□

Correspondance de Galois Nous relions maintenant le domaine des contraintes de changement d'état $\mathbb{FS}^n$ au domaine des fonctions en escalier $\mathcal{FS}^n$ via une correspondance de Galois. Pour la fonction d'abstraction α' (équation (6.10)), nous utilisons le fait que pour toute fonction en escalier finie $f \in \mathcal{FS}^n$, il existe $N \in \mathbb{N}$ et un ensemble d'instants $0 = t_0 < t_1 < \cdots < t_N$ tels que f est constante sur chaque intervalle $[t_i, t_{i+1}[$ et f est constante sur $[t_N, \infty[$. La concrétisation d'une contrainte $\phi = \bigwedge_{0\leq i\leq N} t_i : [x_i]$ est la fonction $f \in \mathcal{FS}^n$ vérifiant $\forall i \in [0, N-1],\ \forall t \in [t_i, t_{i+1}[,\ f(t) = [x_i]$ et $\forall t \geq t_N,\ f(t) = [x_N]$ (fonction γ' donnée par l'équation (6.11)).

$f \in \mathcal{FS}^n$, soit $N \in \mathbb{N}$ et $0 = t_0 \leq \cdots \leq t_N < t_{N+1} = \infty$ tels que f est constante sur $[t_i, t_{i+1}[$.

$$\alpha' : \begin{cases} \mathcal{FS}^n & \to & \mathbb{FS}^n \\ f & \mapsto & \mathit{Norm}(\bigwedge\limits_{0\leq i\leq N} t_i : f(t_i)) \\ \bot_{fsn} & \mapsto & \mathtt{ff}^\sharp \end{cases} \tag{6.10}$$

$$\gamma' : \begin{cases} \mathbb{FS}^n & \to & \mathcal{FS}^n \\ \bigwedge\limits_{0\leq i\leq N} t_i : [x_i] & \mapsto & \lambda t.\,[x_{i_0}] \text{ tel que } i_0 = \max\{i \in [0, N] \ :\ t_i \leq t\} \\ \mathtt{ff}^\sharp & \mapsto & \bot_{fsn} \end{cases} \tag{6.11}$$

Proposition 6.5 *Les fonctions $\alpha' : \mathcal{FS}^n \to \mathbb{FS}^n$ et $\gamma' : \mathbb{FS}^n \to \mathcal{FS}^n$ sont monotones.*

Preuve Soit $f, g \in \mathcal{FS}^n$ telles que $f \sqsubseteq g$, montrons que $\alpha'(f) \Rightarrow^\sharp \alpha'(g)$. Soit $\phi = \alpha'(f)$ et $\psi = \alpha'(g)$, d'après la proposition 6.2, on peut supposer que $\phi = \bigwedge_{0\leq i\leq N} t_i : f(t_i)$ et $\psi = \bigwedge_{0\leq M} u_i : g(u_i)$. Soit donc $i, j \in [0, N] \times [0, M]$ tels que $[t_i, t_{i+1}[\cap[u_j, u_{j+1}[\neq \emptyset$, et soit $t \in [t_i, t_{i+1}[\cap[u_j, u_{j+1}[$. On sait que $f(t) = f(t_i)$ et $g(t) = g(u_j)$, et que $f(t) \sqsubseteq^n g(t)$ car $f \sqsubseteq g$. Donc, $f(t_i) \sqsubseteq^n g(u_j)$, et donc $\phi \Rightarrow^\sharp \psi$. Donc, α' est croissante.
Soit maintenant $\phi, \psi \in \mathbb{FS}^n$ telles que $\phi \Rightarrow^\sharp \psi$, montrons que $\gamma'(\phi) \sqsubseteq \gamma'(\psi)$. On pose $\phi = \bigwedge_{0\leq i\leq N} t_i : [x_i]$ et $\psi = \bigwedge_{0\leq i\leq M} u_i : [y_i]$. Soit $f = \gamma'(\phi)$ et $g = \gamma'(\psi)$, et soit $t \in \mathbb{R}_+$. Soit $i \in [0, N]$ tel que $t \in [t_i, t_{i+1}[$ et $j \in [0, M]$ tel que $t \in [u_j, u_{j+1}[$. On a donc $[t_i, t_{i+1}[\cap[u_j, u_{j+1}[\neq \emptyset$, donc comme

$\phi \Rightarrow^\sharp \psi$, $[x_i] \sqsubseteq^n [y_j]$. Or, $f(t) = [x_i]$ et $g(t) = [y_j]$, donc $f(t) \sqsubseteq^n g(t)$, et donc $f \dot{\sqsubseteq} g$, donc γ' est croissante. □

Proposition 6.6 *Les domaines $\mathcal{FS}^n$ et $\mathbb{FS}^n$ sont reliés par une bijection de Galois :*

$$(\mathcal{FS}^n, \sqsubseteq) \underset{\alpha'}{\overset{\gamma'}{\leftrightarrows}} (\mathbb{FS}^n, \Rightarrow^\sharp) ,$$

c'est-à-dire que (α', γ') forme une correspondance de Galois telle que $\forall f \in \mathcal{FS}^n,\ \gamma' \circ \alpha'(f) = f$ et $\forall \phi \in \mathbb{FS}^n,\ \alpha' \circ \gamma'(\phi) = \phi$.

Preuve Montrons d'abord que (α', γ') est une correspondance de Galois. On sait que α' et γ' sont monotones (proposition 6.5), nous devons montrer qu'elles vérifient l'équation (2.24). Soit donc $f \in \mathcal{FS}^n$ et $\phi \in \mathbb{FS}^n$, et soit $\psi = \alpha'(f)$ et $g = \gamma'(\phi)$. On pose $\phi = \bigwedge_{0 \leq i \leq N} t_i : [x_i]$ et soit $u_0, u_1, \ldots, u_M \in \mathbb{R}_+$ tels que $\alpha'(f) = \bigwedge_{0 \leq i \leq M} u_i : f(u_i)$.
Supposons que $\alpha'(f) \Rightarrow^\sharp \phi$. Soit alors $t \in \mathbb{R}_+$, et soient $i, j \in [0, N] \times [0, M]$ tels que $t \in [t_i, t_{i+1}[$ et $t \in [u_j, u_{j+1}[$. On sait alors que $g(t) = [x_i]$ et $f(t) = f(u_j)$. Comme $[t_i, t_{i+1}[\cap [u_j, u_{j+1}[\neq \emptyset$, on a $f(u_j) \sqsubseteq^n [x_i]$, et donc $f(t) \sqsubseteq^n g(t)$. Donc, on a $f \sqsubseteq g$, soit $f \sqsubseteq \gamma'(\phi)$.
Supposons maintenant que $f \dot{\sqsubseteq} \gamma'(\phi)$. Soit $i, j \in [0, N] \times [0, M]$ tels que $[t_i, t_{i+1}[\cap [u_j, u_{j+1}[\neq \emptyset$, et soit $t \in [t_i, t_{i+1}[\cap [u_j, u_{j+1}[$. On a alors $f(t) = f(u_j)$ et $g(t) = [x_i]$. Comme $f \dot{\sqsubseteq} g$, on a $f(t) \sqsubseteq^n g(t)$, et donc $f(u_j) \sqsubseteq^n [x_i]$. On a donc $\psi \Rightarrow^\sharp \phi$, soit $\alpha'(f) \Rightarrow^\sharp \phi$.
On a donc montré que (α', γ') est une correspondance de Galois. Montrons maintenant l'aspect bijectif. On veut montrer que $\gamma'(\alpha'(f)) = f$. Soit $t \in \mathbb{R}_+$, et $i \in [0, M]$ tel que $t \in [u_i, u_{i+1}[$. Comme $\alpha'(f) = \bigwedge_{0 \leq i \leq M} u_i : f(u_i)$, on sait que $\gamma'(\alpha'(f)) = f(u_i) = f(t)$. Donc, $\gamma'(\alpha'(f)) = f$. De même, on montre que $\alpha'(\gamma'(\phi)) = \phi$. Il suffit en fait de remarquer que γ' est injectif, et d'utiliser le fait que $\gamma' \circ \alpha'$ est la fonction identité. □

On a donc une bijection entre le domaine $\mathcal{FS}^n$ des fonctions en escalier et le domaine $\mathbb{FS}^n$ des contraintes. Pour une contrainte $\phi \in \mathbb{FS}^n$, on peut donc parler de la valeur que prend ϕ au temps t, que nous noterons $\phi(t)$, comme étant la valeur prise par sa concrétisation : $\phi(t) = \gamma'(\phi)(t)$. La bijection de Galois nous indique donc que :

$$\forall \phi, \psi \in \mathbb{FS}^n,\ \phi \Rightarrow^\sharp \psi \Leftrightarrow \gamma'(\phi) \sqsubseteq \gamma'(\psi) \Leftrightarrow \forall t \in \mathbb{R}_+,\ \phi(t) \sqsubseteq^n \psi(t) \tag{6.12}$$

Opérateurs abstraits Nous définissons maintenant les équivalents abstraits $\vee^\sharp$ et $\wedge^\sharp$ des opérateurs d'union $\sqcup$ et d'intersection $\sqcap$ sur le domaine $\mathcal{FS}^n$.

Définition 6.10 (Opérateur d'intersection pour le domaine $\mathbb{FS}^n$) L'opérateur abstrait d'intersection sur $\mathbb{FS}^n$ fonctionne comme décrit dans l'algorithme suivant :

Entrée : $\phi = \bigwedge_{i \in [0,N]} t_i : [x_i] \in \mathbb{FS}^n$
Entrée : $\psi = \bigwedge_{i \in [0,M]} u_i : [y_i] \in \mathbb{FS}^n$
Résultat : $\phi \wedge^\sharp \psi$

début
  si $\phi = \texttt{ff}^\sharp$ ou $\psi = \texttt{ff}^\sharp$ **alors renvoyer** $\texttt{ff}^\sharp$ **finsi**
  pour $0 \leq i \leq N$ **faire**
    $j = \max\{k \ :\ u_k \leq t_i\}$; $[x_i]' = [x_i] \sqcap^n [y_j]$;
    si $[x_i]' = \emptyset$ **alors renvoyer** $\texttt{ff}^\sharp$ **finsi**
  finpour
  pour $0 \leq i \leq M$ **faire**
    $j = \max\{k \ :\ t_k \leq u_i\}$; $[y_i]' = [y_i] \sqcap^n [x_j]$;
    si $[y_i]' = \emptyset$ **alors renvoyer** $\texttt{ff}^\sharp$ **finsi**
  finpour
  renvoyer $Norm\big(\bigwedge_{0 \leq i \leq N} t_i : [x_i]' \wedge \bigwedge_{0 \leq i \leq M} u_i : [y_i]'\big)$;
fin

Intuitivement, l'intersection de $\phi \in \mathbb{FS}^n$ et $\psi \in \mathbb{FS}^n$ fonctionne ainsi : on commence par réduire chaque intervalle intervenant dans chacune des deux contraintes, puis on normalise la conjonction des deux contraintes ainsi obtenues.

Exemple 6.4 La figure 6.6 montre graphiquement l'intersection de deux contraintes : l'intersection des fonctions rouge et bleu (à gauche) donne la fonction grise (à droite).

Proposition 6.7 *L'opérateur $\wedge^\sharp$ est la meilleure abstraction de $\sqcap$, c'est-à-dire que pour toutes contraintes $\phi, \psi \in \mathbb{FS}^n$, on a*

$$\phi \wedge^\sharp \psi = \alpha'\big(\gamma'(\phi) \sqcap \gamma'(\psi)\big) \ . \tag{6.13}$$

Preuve Soit $\phi, \psi \in \mathbb{FS}^n$, avec $\phi \neq \texttt{ff}^\sharp$ et $\psi \neq \texttt{ff}^\sharp$. On pose $\phi = \bigwedge_{0 \leq i \leq N} t_i : [x_i]$ et $\psi = \bigwedge_{0 \leq i \leq M} u_i : [y_i]$. Soit $f = \gamma'(\phi)$ et $g = \gamma'(\psi)$, on montre que $\forall t \in \mathbb{R}_+,\ \gamma'(\phi \wedge^\sharp \psi)(t) = f(t) \sqcap^n g(t)$. Soit $t \in \mathbb{R}_+$, et $i_0 = max\{i \in [0, N] \ : \ t_i \leq t\}$ et $j_0 = max\{i \in [0, M] \ : \ u_i \leq t\}$. D'après la définition de la fonction de normalisation et de $\wedge^\sharp$, on a $\gamma'\big(\phi \wedge^\sharp \psi\big)(t) = [x_{i_0}] \sqcap^n [y_{j_0}]$. Par ailleurs, $f(t) = [x_{i_0}]$ et $g(t) = [y_{j_0}]$. On a donc $\gamma'(\phi \wedge^\sharp \psi) = f \sqcap g$, d'où :

$$\phi \wedge^\sharp \psi = \alpha'\big(f \sqcap g\big) = \alpha'\big(\gamma'(\phi) \sqcap \gamma'(\psi)\big) \ .$$

□

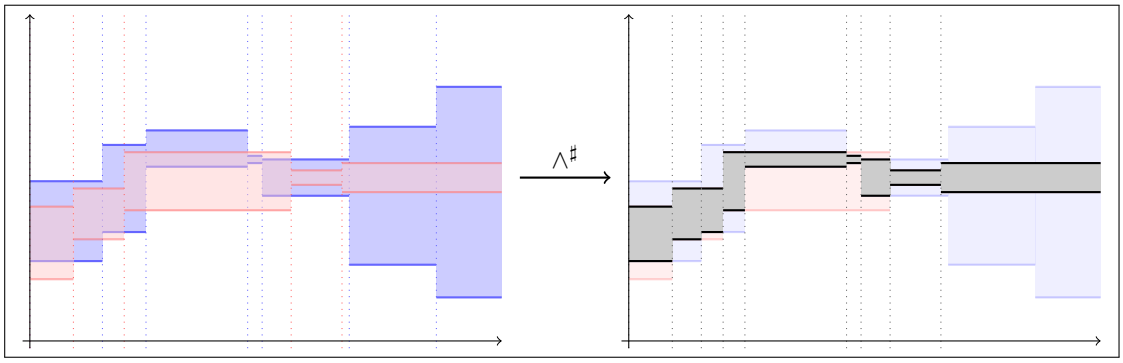


FIG. 6.6 – Opérateur d'intersection pour les contraintes de changement d'état.

Définition 6.11 (Opérateur d'union pour le domaine $\mathbb{FS}^n$) L'opérateur abstrait d'union sur $\mathbb{FS}^n$ fonctionne comme décrit dans l'algorithme suivant :

Entrée : $\phi = \bigwedge_{i \in [0,N]} t_i : [x_i] \in \mathbb{FS}^n$
Entrée : $\psi = \bigwedge_{i \in [0,M]} u_i : [y_i] \in \mathbb{FS}^n$
Entrée : $\phi \vee^\sharp \psi$

début
  si $\phi = \texttt{ff}^\sharp$ **alors renvoyer** ψ **finsi**
  si $\psi = \texttt{ff}^\sharp$ **alors renvoyer** ϕ **finsi**
  pour $0 \leq i \leq N$ **faire**
    $j = \max\{k \ : \ u_k \leq t_i\}$; $[x_i]' = [x_i] \sqcup^n [y_j]$;
  finpour
  pour $0 \leq i \leq M$ **faire**
    $j = \max\{k \ : \ t_k \leq u_i\}$; $[y_i]' = [y_i] \sqcup^n [x_j]$;
  finpour
  renvoyer $Norm\big(\bigwedge_{0 \leq i \leq N} t_i : [x_i]' \wedge \bigwedge_{0 \leq i \leq M} u_i : [y_i]'\big)$;
fin

Intuitivement, pour effectuer l'union de deux contraintes ϕ et ψ, on effectue la même opération que pour l'intersection sauf que l'on commence par faire l'union des intervalles de ϕ et ψ avant de normaliser la conjonction des contraintes.

Exemple 6.5 La figure 6.7 montre l'effet de l'opérateur d'union entre contraintes : l'intersection des fonctions rouge et bleu (à gauche) donne la fonction grise (à droite).

Proposition 6.8 *L'opérateur $\vee^\sharp$ est la meilleure abstraction de $\sqcup$, c'est-à-dire que pour toutes contraintes $\phi, \psi \in \mathbb{FS}^n$, on a*

$$\phi \vee^\sharp \psi = \alpha'\big(\gamma'(\phi) \sqcup \gamma'(\psi)\big) \ . \tag{6.14}$$

Preuve La preuve est similaire à celle de la proposition 6.7. □

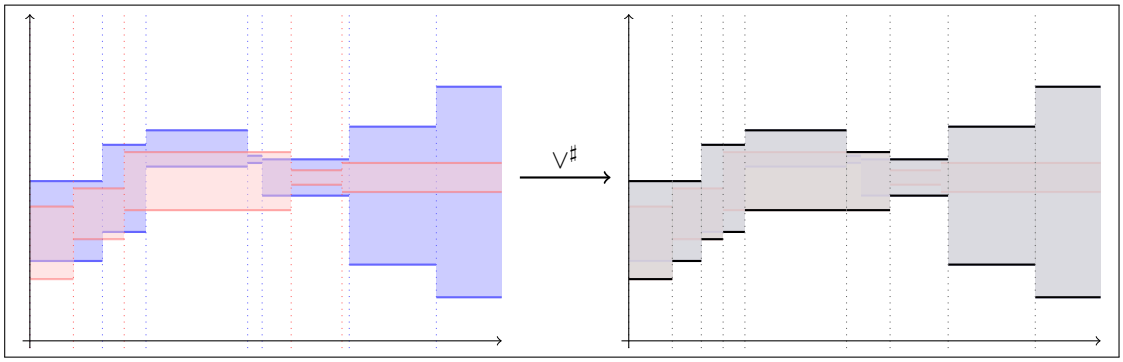


FIG. 6.7 – Opérateur d'union pour les contraintes de changement d'état.

Théorème 6.9 $\big(\mathbb{FS}^n, \Rightarrow^\sharp, \wedge^\sharp, \vee^\sharp, \texttt{ff}^\sharp, \texttt{tt}^\sharp\big)$ *est un treillis.*

Preuve Il faut montrer que $\vee^\sharp$ calcule bien la borne supérieure et $\wedge^\sharp$ la borne inférieure. Soit $\phi, \psi \in \mathbb{FS}^n$. Si on montre que $\gamma(\phi \vee^\sharp \psi) = \gamma(\phi) \sqcup \gamma(\psi)$, alors comme $\mathbb{FS}^n$ et $\mathcal{FS}^n$ sont reliés par une bijection de Galois, on pourra conclure que $\vee^\sharp$ calcule la borne supérieure. De même, si on montre que $\gamma(\phi \wedge^\sharp \psi) = \gamma(\phi) \sqcap \gamma(\psi)$, on pourra conclure que $\wedge$ calcule la borne inférieure. Les propositions 6.7 et 6.8 permettent justement de conclure que ces conditions sont vérifiées. □

Remarque Pour les mêmes raisons que pour $\mathcal{FS}^n$, le treillis $\mathbb{FS}^n$ n'est ni un treillis complet ni un CPO.

Les éléments de $\mathbb{FS}^n$ sont facilement et efficacement représentables en machine. De plus, les fonctions de comparaison, union et intersection sont efficaces : la normalisation, qui est l'opération la plus coûteuse, s'exécute en $\mathcal{O}\big(n \cdot log(n)\big)$ car elle nécessite un tri des contraintes présentes. On utilisera donc ce domaine comme abstraction des ensembles de fonctions continues. Il faut donc que l'on adapte le critère CS1 (définition 6.6) au domaine $\mathbb{FS}^n$ pour s'assurer de la sûreté des sémantiques abstraites.

Définition 6.12 (Critère de sûreté pour la sémantique abstraite (2)) Une sémantique abstraite $(\!|.|\!)^\sharp : \mathcal{PCI} \to \mathbb{FS}^n$ vérifie le critère de sûreté CS2 si et seulement si, pour tout PCI P, elle vérifie la formule (6.15), avec $\phi = (\!|P|\!)^\sharp = \bigwedge_{0 \le i \le N} t_i : [x_i]$ et $t_{N+1} = \infty$.

$$\forall i \in [0, N],\ \forall t \in [t_i, t_{i+1}[,\ \underline{[x_i]} \le inf\{y(t)\ :\ y \in (\!|P|\!)\} \quad \text{et} \quad sup\{y(t)\ :\ y \in (\!|P|\!)\} \le \overline{[x_i]}\ . \tag{6.15}$$

Théorème 6.10 *Si $(\!|.|\!)^\sharp$ vérifie le critère CS2, alors pour tout PCI P, on a :*

$$(\!|P|\!) \subseteq \gamma \circ \gamma'\left((\!|P|\!)^\sharp\right)$$

Preuve On montre que si $(\!|.|\!)^\sharp$ vérifie CS2, alors $\gamma' \circ (\!|.|\!)^\sharp$ vérifie CS1. Ceci est évident d'après la définition de γ' (définition 6.5). Le théorème 6.1 s'applique donc sur $\gamma' \circ (\!|.|\!)^\sharp$, ce qui prouve le théorème 6.10. □

6.3 Calcul de la sémantique abstraite

Nous allons donc utiliser le domaine des contraintes $\mathbb{FS}^n$ comme domaine abstrait, et nous cherchons une sémantique abstraite $(\!|.|\!)^\sharp : \mathcal{PCI} \to \mathbb{FS}^n$ qui vérifie le critère CS2. Cette sémantique abstraite nous est fournie par les algorithmes d'intégration garantie décrits au chapitre 3.

6.3.1 L'intégration garantie est une sémantique abstraite valide

Pour construire la sémantique abstraite, nous allons simplement utiliser les algorithmes d'intégration garantie de la manière suivante : la première phase de l'algorithme (encadrement a priori, voir la section 3.3.2) nous donne, à partir d'un PCI $P := \left(\dot{y} = F(y, [k]\,),\ y(0) \in [y_0]\,\right)$, un pas d'intégration h et un encadrement $[y]$ tels que $\forall t \in [0, h]\,,\ \forall y \in (\!|P|\!),\ y(t) \in [y]$ (on pourra trouver les preuves dans [BM07a, NJC99]). De plus, la seconde phase de l'intégration garantie (section 3.3.3) donne $[y_h]$ tel que $\forall y \in (\!|P|\!),\ y(h) \in [y_h]$. Le théorème suivant est donc vrai :

Théorème 6.11 (Fonction d'intégration garantie) *Il existe une fonction `IG` qui, étant donné un PCI $P \in \mathcal{PCI}$, fournit $h \in \mathbb{R}_+$, $[y]\,, [y_h] \in \mathbb{IR}^n$ tels que :*

1. $\forall y \in (\!|P|\!),\ \forall t \in [0, h]\,,\ y(t) \in [y]$ *et*
2. $\forall y \in (\!|P|\!),\ y(h) \in [y_h]$.

Si la fonction renvoie $h = 0$, cela voudra dire que l'intégration garantie a échoué.

Définition 6.13 (Sémantique abstraite) La sémantique abstraite d'un PCI $P \in \mathcal{PCI}$ est définie comme le résultat de l'algorithme suivant :

Entrée : $P = \left(\dot{y} = F(y, [k]),\ y(0) \in [y_0]\,\right) \in \mathcal{PCI}$
Entrée : $N \in \mathbb{N}^*$; `/* nombre de pas à effectuer */`
Sortie : $(\!|P|\!)^\sharp_N$
début
  $(h, [y]\,, [y_h]) = \texttt{IG}(P)$;
  si $h = 0$ **alors renvoyer** $\texttt{tt}^\sharp$; **finsi**
  $res = 0 : [y]\,;\ i = 1$;
  tant que $i < N$ **faire**
    $P = \left(\dot{y} = F(y, [k]),\ y(0) \in [y_h]\,\right)$;
    $(h, [y]\,, [y_h]) = \texttt{IG}(P)$;
    si $h = 0$ **alors renvoyer** $res \wedge t : \top^n_I$; **finsi**
    $res = res \wedge t : [y]$;
    $i = i + 1$; $t = t + h$;
  fintantque
  renvoyer $res \wedge t : \top^n_I$;
fin

Pour un PCI $P \in \mathcal{PCI}$ et un entier $N \in \mathbb{N}^*$, on notera $(\!|P|\!)^\sharp_N$ le résultat de l'algorithme.

La sémantique abstraite utilise donc un algorithme d'intégration garantie pour calculer une sur-approximation des solutions du PCI par une fonction en escalier à N marches. Le paramètre N permet de contrôler la précision de l'abstraction : plus N est grand, plus l'abstraction est précise. La sûreté de la sémantique abstraite ainsi calculée est prouvée par le théorème suivant.

Théorème 6.12 (Sûreté de la sémantique abstraite.) *Pour tout $N \in \mathbb{N}^*$, la fonction $(\!|.|\!)^\sharp_N$ donné à la définition 6.13 vérifie le critère CS2.*

Preuve On prouve le théorème par récurrence sur N. Si $N = 1$, la preuve est évidente d'après le théorème 6.11. Soit maintenant $N \geq 1$ tel que la fonction $(\!|.|\!)^\sharp_N$ vérifie le critère CS2, nous devons montrer que $(\!|.|\!)^\sharp_{N+1}$ le vérifie également. Soit donc $P \in \mathcal{PCI}$ un problème de Cauchy intervalle et soit $\phi = (\!|P|\!)^\sharp_{N+1}$. D'après la définition 6.13, ϕ est construite ainsi : on calcule $\psi = (\!|P|\!)^\sharp_N$, puis on effectue un dernier pas d'intégration garanti pour la dernière contrainte. D'après l'hypothèse de récurrence, ψ vérifie l'équation (6.15), et d'après la construction de la dernière contrainte et le théorème 6.11, ϕ vérifie également cette équation pour $i = N + 1$. Donc, ϕ vérifie l'équation (6.15), donc $(\!|.|\!)^\sharp_{N+1}$ vérifie le critère CS2. □

6.3.2 Motivation pour une nouvelle méthode d'intégration garantie

Les algorithmes classiques d'intégration garantie (AWA ou VNODE notamment) utilisent, pour résumer, un développement en séries de Taylor de la solution de l'EDO à intégrer, qu'ils transforment en méthode de calcul garanti en utilisant une arithmétique d'intervalles (voir chapitre 3). La formule ainsi obtenue est ensuite réécrite pour obtenir une méthode la plus stable possible. Cette réécriture consiste notamment à remplacer le calcul par intervalle classique en un calcul par forme centrée, ce qui revient à calculer l'encadrement comme un point associé à une erreur, soit le diamètre de l'intervalle. Les méthodes les plus stables (comme COSY-VI) vont encore plus loin en représentant les termes intervalles par des formes de Taylor, ce qui réduit le wrapping effect mais augmente largement le temps de calcul. Les méthodes classiques à base d'intervalles, si elles sont relativement efficaces, souffrent de plusieurs problèmes. Tout d'abord, pour des problèmes non linéaires, les résultats tendent à être fortement surapproximés à cause du wrapping effect qui rend les algorithmes instables. Pour les problèmes les plus complexes, cette instabilité est couplée à une baisse d'efficacité. En effet, il devient alors important d'utiliser un ordre élevé dans le développement en série de Taylor. Presque toutes les implémentations existantes utilisent des algorithmes de différentiation automatique pour cela, ce qui revient à recalculer à chaque pas les dérivées successives de la fonction à intégrer. Cette opération peut être très coûteuse en temps. Enfin, le principal reproche que l'on pourrait faire aux méthodes classiques d'intégration garantie est leur éloignement des méthodes d'intégration numérique. En effet, les numériciens ont développé de nombreuses méthodes d'approximation des solutions des équations différentielles et possèdent souvent une grande expertise sur ces méthodes. Par ailleurs, ces méthodes sont très souvent utilisées lors du test des systèmes critiques embarqués pour simuler l'environnement extérieur (c'est notamment ce que nous avons fait pour la simulation du problème des deux réservoirs en Simulink, annexe A). Une méthode d'intégration garantie qui serait construite sur une méthode numérique permettrait donc non seulement d'utiliser l'expertise acquise sur les méthodes numériques pour choisir finement le pas d'intégration et rendre la méthode la plus stable possible, mais surtout donnerait une information supplémentaire qui est la distance entre la simulation numérique et la vraie solution. Dans cette optique, on peut voir le programme de résolution numérique comme l'implémentation, suivant un algorithme particulier, du problème de la résolution de l'équation différentielle. Mesurer la qualité de cette implémentation revient à mesurer la distance entre les résultats fournis par l'intégration numérique et les résultat attendus, c'est-à-dire la solution de l'équation différentielle.

Nous avons développé une méthode d'intégration garantie nommée GRKLib qui repose sur ces idées, c'est-à-dire qu'elle s'appuie sur une méthode numérique classique (en l'occurrence une méthode de Runge-Kutta, mais les idées que nous présentons peuvent s'appliquer dans un cadre bien plus général). Notre ambition en commençant ce travail sur un algorithme de Runge-Kutta

garantie était très élevée : nous pensions pouvoir limiter le wrapping effect en séparant, dès le début, calcul flottant et calcul par intervalle. Cette technique a été appliquée avec succès pour l'analyse des programmes numériques (domaine de l'erreur globale, [Mar05]) et nous pensons qu'elle peut aider à accroître la stabilité des méthodes d'intégration garantie. De plus, nous avons choisi une implémentation différente des techniques existantes qui n'est pas fondée sur la différentiation automatique, ce qui permet d'avoir une implémentation efficace grâce à un temps de calcul par pas d'intégration faible. Il faut toutefois signaler que les résultats obtenus n'ont pas été à la hauteur de nos espérances, les performances de GRKLib restent en effet en deçà des performances de VNODE, mais notre travail a permis de prouver la faisabilité et l'intérêt de cette approche qui consiste à utiliser les techniques d'intégration numérique classique pour faire de l'intégration garantie.

6.4 GRKLib en détail

Nous expliquons maintenant comment fonctionne la méthode GRKLib. Nous nous intéressons donc à une EDI de dimension n, $\dot{y} = F(y)$, avec une condition initiale donnée sous la forme $y(0) \in y_0 + [\epsilon_0]$, où $y_0 \in \mathbb{R}^n$ est un point et $[\epsilon_0] \in I\!R^n$ représente l'erreur (ou incertitude) initiale. Nous noterons y_∞ une solution quelconque du problème de Cauchy $\dot{y} = F(y),\ y(0) = y'_0$, avec $y'_0 \in y_0 + [\epsilon_0]$.

6.4.1 Principe de l'algorithme de Runge-Kutta garanti

La méthode GRKLib est construite à partir d'un algorithme d'intégration numérique de Runge-Kutta d'ordre 4, la méthode RK4. Les algorithmes de Runge-Kutta sont des schémas d'intégration implicites qui calculent une suite de points $(y_n)_{n\in\mathbb{N}}$ et une suite d'instants $(t_n)_{n\in\mathbb{N}}$, tels que pour tout $n \in \mathbb{N}$, y_n est une approximation de la valeur de la solution de l'EDO à $t = t_n$. Pour obtenir y_{n+1}, on commence par choisir un pas d'intégration h_n, puis on calcule y_{n+1} en n'utilisant que y_n et des points intermédiaires. En cela, la méthode RK4 peut être vue comme une extension de la méthode d'Euler ($y_{n+1} = y_n + h_n \cdot F(y_n)$) et de la méthode du milieu ($y_{n+1} = y_n + h_n \cdot F\big(y_n + h_n/2 \cdot F(y_n)\big)$). La méthode RK4 utilise quatre points intermédiaires, donnés par la formule (6.16).

$$\begin{array}{rcl} k_1 & = & F(y_n) \\ k_2 & = & F(y_n + h_n/2 \cdot k_1) \\ k_3 & = & F(y_n + h_n/2 \cdot k_2) \\ k_4 & = & F(y_n + h_n \cdot k_3) \\ y_{n+1} & = & y_n + \frac{h_n}{6}\,(k_1 + 2k_2 + 2k_3 + k_4) \end{array} \tag{6.16}$$

Il est bien connu [SB93] que cette méthode est d'ordre 4, ce qui veut dire que la différence entre la valeur exacte $y_\infty(t_n + h_n)$ et y_{n+1} est censée être proportionnelle à h_n^5. Si ceci est vrai pour le premier pas d'intégration, nous verrons que l'erreur grandit à mesure que l'on applique la formule (6.16) pour obtenir plus de points d'approximation. Nous montrerons comment on peut mesurer cet accroissement de l'erreur pour calculer des bornes garanties sur l'erreur entre y_n et $y_\infty(t_n)$ pour tout $n \in \mathbb{N}$. Pour cela, nous définissons la fonction d'itération Φ qui dépend de y et h de telle sorte que l'on ait $y_{n+1} = \Phi(y_n, h_n)$.

$$\begin{array}{rcl} k_1(y,h) & = & F(y) \\ k_2(y,h) & = & F(y + h/2 \cdot k_1(y,h)) \\ k_3(y,h) & = & F(y + h/2 \cdot k_2(y,h)) \\ k_4(y,h) & = & F(y + h \cdot k_3(y,h)) \\ \Phi(y,h) & = & y + \frac{h}{6}\big(k_1 + 2k_2 + 2k_3 + k_4\big)(y,h) \end{array} \tag{6.17}$$

Nous définissons également les deux fonctions ψ_n et ϕ_n à chaque étape, qui nous serons utiles pour calculer les termes d'erreur. La fonction ψ_n calcule, à pas h_n constant, la position de y_{n+1} en fonction de y_n, alors que ϕ_n calcule, en supposant que $y_n = y(t_n)$, la position de y_{n+1} en fonction du pas d'intégration.

$$\psi_n : \left\{ \begin{array}{rcl} \mathbb{R}^n & \to & \mathbb{R}^n \\ y & \mapsto & \Phi(y, h_n) \end{array} \right. \qquad \phi_n : \left\{ \begin{array}{rcl} \mathbb{R}_+ & \to & \mathbb{R}^n \\ t & \mapsto & \Phi\big(y(t_n), t - t_n\big). \end{array} \right. \tag{6.18}$$

La méthode GRKLib fonctionne alors ainsi : si on connaît à l'instant t_n un point d'approximation (un nombre à virgule flottante) $\boldsymbol{y_n}$ et l'erreur commise $[\epsilon_n]$ de telle sorte que $y_\infty(t_n) \in \boldsymbol{y_n} + [\epsilon_n]$, on choisit un pas d'intégration h_n puis on calcule le prochain point d'approximation $\boldsymbol{y_{n+1}} = \Phi(\boldsymbol{y_n}, h_n)$ et en même temps une surapproximation $[\epsilon_{n+1}]$ de l'erreur $y_\infty(t_{n+1}) - \boldsymbol{y_{n+1}}$.

6.4.2 Trois sources d'erreurs

L'erreur à l'étape $n+1$ est décomposée en trois termes représentés graphiquement par la figure 6.8. Le point $\boldsymbol{y_{n+1}}$ est obtenu en appliquant la formule (6.16) en utilisant des nombres à virgule flottante. Notons y^r_{n+1} la valeur réelle obtenue en utilisant la même formule mais avec des nombres réels. Les règles de calcul données par la norme IEE754, et qui sont spécifiques à la représentation finie des nombres flottants, créent une légère différence entre $\boldsymbol{y_{n+1}}$ et y^r_{n+1} que nous appellerons "erreur de calcul". Cette première source d'erreur sera calculée en utilisant le domaine de l'erreur globale (section 6.4.5). Par ailleurs, la fonction à intégrer va propager l'erreur associée au pas précédent. Soit y^*_{n+1} le point (réel) que nous aurions obtenu en appliquant les formules de RK4 (équation (6.16)) en prenant comme point initial $y(t_n)$ à la place de $\boldsymbol{y_n}$, autrement dit $y^*_{n+1} = \Phi(y(t_n), h_n)$. La distance entre y^*_{n+1} et y^r_{n+1} représente la propagation de l'erreur initiale ϵ_n lors du calcul de y_{n+1}. Enfin, la méthode de RK4 introduit elle même une erreur car elle n'est que d'ordre 4. Cette erreur est la distance entre $y_\infty(t_{n+1})$, la valeur réelle que l'on cherche, et y^*_{n+1}, la valeur réelle que l'on obtiendrait en appliquant RK4 depuis $y(t_n)$. L'erreur globale ϵ_{n+1} est donc calculée comme la somme de trois termes : $\epsilon_{n+1} = e_{n+1} + \chi_{n+1} + \eta_{n+1}$ avec :

- η_{n+1} est l'erreur "sur un pas" introduite par la méthode RK4 : $\eta_{n+1} = y_\infty(t_{n+1}) - y^*_{n+1}$,
- χ_{n+1} est la propagation, par le système dynamique, de l'erreur précédente : $\chi_{n+1} = y^*_{n+1} - y^r_{n+1}$,
- e_{n+1} est l'erreur de calcul due à la précision finie des calculs : $\epsilon_{n+1} = y^r_{n+1} - \boldsymbol{y_{n+1}}$.

Nous montrons maintenant comment calculer chacun de ces termes.

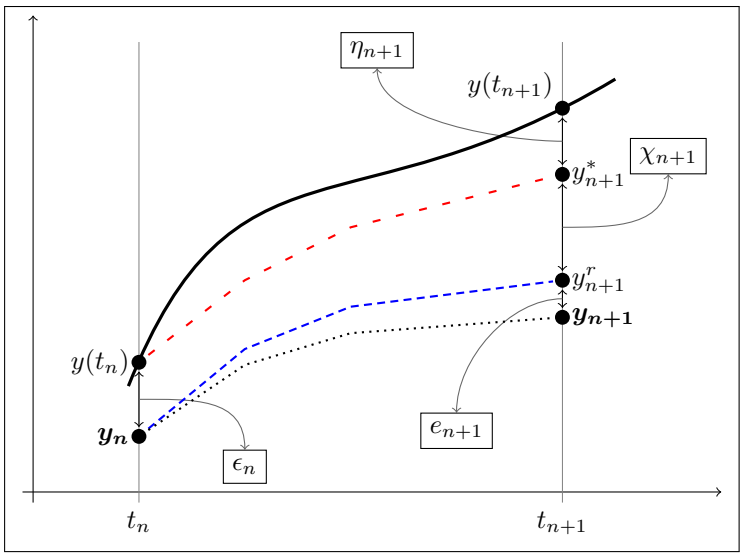


FIG. 6.8 – Trois types d'erreurs : l'erreur sur un pas η_{n+1}, l'erreur de calcul e_{n+1} et l'erreur propagée χ_{n+1}.

6.4.3 Erreur sur un pas

L'erreur sur un pas apparaît car la trajectoire de la solution de l'équation différentielle (courbe pleine noire sur la figure 6.8) s'écarte de la trajectoire donnée par les formules de RK4 (courbe en pointillés rouges). On sait que $y^*_{n+1} = \phi_n(t_n)$, donc $\eta_{n+1} = y_\infty(t_{n+1}) - \phi(t_{n+1})$ mesure la distance, en t_{n+1} entre les fonctions y_∞ et ϕ. Ces deux fonctions sont égales en $t = t_n$ et vérifient de plus la proposition 6.13.

Proposition 6.13 *Les quatre premières dérivées de y_∞ et ϕ_n sont égales en $t = t_n$:*

$$\forall i \in [0,4],\ \frac{d^i y}{dt^i}(t_n) = \frac{d^i \phi_n}{dt^i}(t_n).$$

Preuve La preuve de cette proposition est initialement apportée par [Bie51], nous en donnons une quelque peu différente à l'annexe B. □

En utilisant la proposition 6.13 et la décomposition en série de Taylor de y_∞ avec reste de Lagrange, on obtient la proposition suivante.

Proposition 6.14 *Il existe $\tau \in [t_n, t_{n+1}]$ tel que :*

$$\eta_{n+1} = y_\infty(t_{n+1}) - \phi_n(t_{n+1}) = \frac{h_n^5}{120} \frac{d^5(y_\infty - \phi_n)}{dt^5}(\tau).$$

Si on essaie de calculer la dérivée cinquième intervenant dans la proposition 6.14, on obtient :

$$\frac{d^5(y_\infty - \phi_n)}{dt^5} = \frac{d^5(y_\infty)}{dt^5} - \frac{d^5(\phi_n)}{dt^5} = \frac{d^4(F)}{dt^4} - \frac{d^5(\phi_n)}{dt^5}$$

et donc

$$\eta_{n+1} = \frac{h_n^5}{120} \left(\frac{d^4(f)}{dt^4}\big(y_\infty(\tau)\big) - \frac{d^5(\phi_n)}{dt^5}(\tau) \right) .$$

Il nous faut donc maintenant encadrer chacun des termes $\frac{d^4(F)}{dt^4}\big(y_\infty(\tau)\big)$ et $\frac{d^5(\phi_n)}{dt^5}(\tau)$. La fonction ϕ_n est une fonction d'une seule variable réelle, t, donc sa dérivée cinquième également. On peut donc simplement construire une fonction d'inclusion pour $\frac{d^5(\phi_n)}{dt^5}$ et on a alors

$$\frac{d^5(\phi_n)}{dt^5}(\tau) \in \frac{d^5(\phi_n)}{dt^5}\big([t_n, t_{n+1}]\big). \tag{6.19}$$

Pour le deuxième terme, l'encadrement est plus compliqué à trouver. En effet, pour encadrer $\frac{d^4(F)}{dt^4}\big(y_\infty(\tau)\big)$, il faut encadrer $y_\infty(\tau)$ puis utiliser une fonction d'inclusion pour $\frac{d^4(F)}{dt^4}$. Il nous faut donc un encadrement de $y_\infty(\tau)$, où τ est inconnu dans $[t_n, t_{n+1}]$. Autrement dit, nous devons calculer un encadrement a priori $[\widetilde{y_n}]$ qui vérifie $\forall t \in [t_n, t_{n+1}]$, $y_\infty(t) \in [\widetilde{y_n}]$. Pour cela, nous allons utiliser une technique similaire à la technique de l'encadrement constant présenté au chapitre 3, et que nous rappelons brièvement ici. Notons $[y_n] = y_n + [\epsilon_n]$. La méthode de l'encadrement constant revient à trouver un post point fixe de l'opérateur $\Psi : I^n_{\mathbb{R}} \to I^n_{\mathbb{R}}$ défini par $\Psi\big([R]\big) = [y_n] + [0, h_n] \cdot F([R])$. Le théorème 3.6 nous assure que si $[R]$ est un post point fixe de Ψ (c'est-à-dire que $\Psi([R]) \subseteq [R]$), alors y_∞ vérifie $\forall t \in [t_n, t_{n+1}]$, $y_\infty(t) \in [R]$. Remarquons que Ψ est une fonction continue sur les treillis $I^n_{\mathbb{R}}$, donc elle possède un plus petit point fixe lfp_Ψ (et par conséquent un plus petit post point fixe). Le théorème de Kleene 2.3 nous permet même de le calculer en itérant la fonction Ψ à partir du plus petit élément. Cependant, il est clair que $[y_n] \subseteq lfp_\Psi$, de sorte qu'on peut commencer l'itération à $[y_n]$. Généralement, les méthodes d'intégration garantie arrêtent l'itération au bout de la première étape si le post point fixe n'est pas trouvé, et choisissent alors un pas plus petit. Nous avons choisi de garder le même pas mais de poursuivre l'itération de Kleene en effectuant à chaque étape un élargissement : si $[R]' = \Psi([R]) \subsetneq [R]$, on agrandit $[R]'$ d'un facteur multiplicatif α, c'est-à-dire on pose $[R]'' = m\big([R]'\big) + \alpha \times r\big([R]'\big) \times [-1, 1]$ où $m\big([R]'\big)$ est le milieu de $[R]'$ et $r\big([R]'\big)$ son rayon. On recommence ensuite l'itération depuis $[R]''$ et on s'arrête lorsque l'on trouve $[R]$ tel que $\Psi([R]) \subseteq [R]$. L'expérience montre qu'en moyenne, il suffit de 3 itérations pour obtenir un post point fixe.

On utilise alors $[\widetilde{y_n}] = \Psi([R])$ comme encadrement a priori de y_∞ sur l'intervalle $[t_n, t_{n+1}]$, et on a donc une surapproximation de η_{n+1} donnée par :

$$\eta_{n+1} \in \frac{h^5}{120} \cdot \left(\frac{d^4 f}{dt^4}\big([\widetilde{y_n}] \big) - \frac{d^5(\phi_n)}{dt^5}\big([t_n, t_{n+1}]\big) \right) \tag{6.20}$$

6.4.4 Propagation de l'erreur

Nous nous intéressons maintenant au deuxième terme d'erreur qui provient de ce que l'erreur du pas précédent ϵ_n est transportée au pas $n+1$ par le système dynamique. En effet, lorsqu'on calcule y^r_{n+1}, on applique en fait la méthode RK4 pour approcher la solution de l'EDO passant par $\boldsymbol{y_n}$ en t_n, alors que la vraie solution passe par $y(t_n)$ en t_n. On a donc $y^*_{n+1} = \psi_n(y(t_n))$ et $y^r_{n+1} = \psi_n(\boldsymbol{y_n})$. L'erreur χ_{n+1} est donc la distance entre les valeurs prises par $\psi_n : \mathbb{R}^n \to \mathbb{R}^n$ en deux points différents. Cette distance se mesure en utilisant la matrice jacobienne de ψ_n. Rappelons que pour toute fonction $f : \mathbb{R}^n \to \mathbb{R}^n$, $f = (f_1, \ldots, f_n)$ avec pour tout $i \in [0, n]$ $f_i : \mathbb{R}^n \to \mathbb{R}$, la matrice jacobienne $J(f)$ est la matrice des dérivées partielles :

$$J(f) = \begin{pmatrix} \frac{\partial f_1}{\partial x_1} & \frac{\partial f_1}{\partial x_2} & \cdots & \frac{\partial f_1}{\partial x_d} \\ \frac{\partial f_2}{\partial x_1} & \frac{\partial f_2}{\partial x_2} & \cdots & \frac{\partial f_2}{\partial x_d} \\ & \cdots & & \\ \frac{\partial f_d}{\partial x_1} & \frac{\partial f_d}{\partial x_2} & \cdots & \frac{\partial f_d}{\partial x_d} \end{pmatrix} . \tag{6.21}$$

Nous noterons également $J(f, Y)$ la matrice obtenue en appliquant chacune de ces dérivées partielles au point $Y \in \mathbb{R}^n$:

$$J(f, Y) = \begin{pmatrix} \frac{\partial f_1}{\partial x_1}(Y) & \frac{\partial f_1}{\partial x_2}(Y) & \cdots & \frac{\partial f_1}{\partial x_d}(Y) \\ \frac{\partial f_2}{\partial x_1}(Y) & \frac{\partial f_2}{\partial x_2}(Y) & \cdots & \frac{\partial f_2}{\partial x_d}(Y) \\ & \cdots & & \\ \frac{\partial f_d}{\partial x_1}(Y) & \frac{\partial f_d}{\partial x_2}(Y) & \cdots & \frac{\partial f_d}{\partial x_d}(Y) \end{pmatrix} .$$

Nous rappelons également le théorème des valeurs intermédiaires appliqué aux fonctions multivariées, qui sert à mesurer la distance entre $\psi_n(\boldsymbol{y_n})$ et $\psi_n(y_\infty(t_n))$.

Théorème 6.15 (Théorème des valeurs intermédiaires) *Soit $D \subseteq \mathbb{R}^n$ un ouvert de $\mathbb{R}^n$, et soit $f : D \to \mathbb{R}^n$ une fonction différentiable sur D. Alors, pour tout $a \in D$ et $u \in \mathbb{R}^n$ tel que $[a, a+u] \subseteq D$, il existe $\theta \in]0,1[$ tel que*

$$f(a+u) - f(a) = J(f, a + \theta \cdot u) \cdot u.$$

La fonction ψ_n est clairement différentiable sur $\mathbb{R}^n$, donc en appliquant le théorème 6.15, on sait qu'il existe $\theta \in]0,1[$ tel que, si on pose $c = \boldsymbol{y_n} + \theta \cdot (y(t_n) - \boldsymbol{y_n})$, on a :

$$\chi_{n+1} = \psi_n(y(t_n)) - \psi_n(y_n) = J\big(\psi_n, c\big) \cdot \big(y(t_n) - \boldsymbol{y_n}\big) .$$

On peut maintenant utiliser cette formule pour calculer une surapproximation de χ_{n+1}. Soit $[y'_n]$ qui contient $\boldsymbol{y_n}$ et $\boldsymbol{y_n} + [\epsilon_n]$, par exemple $[y'_n] = [\boldsymbol{y_n}, \boldsymbol{y_n}] \sqcup (\boldsymbol{y_n} + [\epsilon_n])$. On sait alors que $\forall \theta \in]0,1[,\ \boldsymbol{y_n} + \theta \cdot (y(t_n) - \boldsymbol{y_n}) \in [y'_n]$. Par ailleurs, on a $y(t_n) - \boldsymbol{y_n} \in [\epsilon_n]$. On obtient donc la formule suivante pour l'encadrement de l'erreur χ_{n+1} :

$$\chi_{n+1} \quad \in \quad J\big(\psi_n, \big[y'_n\big]\big) \cdot [\epsilon_n] \ . \tag{6.22}$$

La formule (6.22) donne une méthode pour calculer χ_{n+1}, mais une évaluation naïve en utilisant une arithmétique d'intervalle donne de mauvais résultats à cause du wrapping effect. Dans l'implémentation de GRKLib, nous utilisons la méthode de décomposition QR de Löhner (voir chapitre 3) pour le limiter.

6.4.5 Erreur de calcul flottant

L' erreur de calcul flottant est liée au calcul sur ordinateur du point d'approximation $\boldsymbol{y_{n+1}}$. Cette erreur correspond à la différence entre les calcul effectués avec une précision finie et qui mène au point $\boldsymbol{y_{n+1}}$, et les calculs que l'on effectuerait avec une précision infinie et qui mèneraient au point y^r_{n+1}. Cette différence tient à l'implémentation des calculs sur les nombres à virgule flottante, définis par la norme IEEE754. Cette erreur peut être réduite en utilisant une arithmétique dite de "multi-précision" comme la bibliothèque MPFR [DHL+05], mais ne peut être éliminée et doit donc être prise en compte. Il faut donc calculer, en même temps que le nombre flottant $\boldsymbol{y_{n+1}}$, un intervalle $[\boldsymbol{e_{n+1}}]$ tel que $y^r_{n+1} \in \boldsymbol{y_{n+1}} + [\boldsymbol{e_{n+1}}]$. L'intervalle $[\boldsymbol{e_{n+1}}]$ doit donc surapproximer *toutes* les erreurs de calcul qui apparaissent lors du calcul de $\boldsymbol{y_{n+1}}$. Pour cela, nous utilisons l'arithmétique de l'*erreur globale*, qui a été initialement introduite pour l'analyse statique de la précision numérique des programmes C [PGM03]. Nous en donnons ici une description assez rapide, on pourra trouver plus de détails et une comparaison avec les autres domaines utilisés pour la validation de codes numériques dans [Mar05].

L'idée du domaine de l'erreur globale est d'attacher à chaque nombre à virgule flottante $\boldsymbol{f_a}$ un nombre réel qui mesure la distance entre $\boldsymbol{f_a}$ et le nombre réel a que $\boldsymbol{f_a}$ est censé représenter. Ainsi, un nombre réel a avec erreur globale peut s'écrire comme :

$$a = \boldsymbol{f_a}\vec{\epsilon_f} + e_a\vec{\epsilon_e}, \quad \text{avec } \boldsymbol{f_a} \in \mathbb{F} \text{ et } e_a \in \mathbb{R}\ . \tag{6.23}$$

Dans l'équation (6.23), les termes $\vec{\epsilon_f}$ et $\vec{\epsilon_e}$ sont des variables formelles permettant de séparer le nombre flottant $\boldsymbol{f_a}$ de l'erreur e_a. Le terme e_a est la distance entre $\boldsymbol{f_a}$ et le réel qu'il représente, on doit donc avoir $e_a \in \mathbb{R}$. Pour une implémentation, on choisira bien sûr un intervalle $[\boldsymbol{e_a}]$ pour représenter e_a. La représentation de l'équation (6.23) signifie que a est un nombre flottant dont la valeur est $\boldsymbol{f_a}$ et qui représente un nombre réel x_a tel que $x_a \in \boldsymbol{f_a} + [\boldsymbol{e_a}]$. L'avantage principal de cette représentation est que l'encadrement final de x_a (c'est-à-dire $\boldsymbol{f_a} + [\boldsymbol{e_a}]$) ne contient pas nécessairement le point flottant $\boldsymbol{f_a}$, alors que toutes les représentations à base d'intervalle (que ce soit une représentation minimum/maximum ou une représentation milieu/rayon) donne des encadrements qui contiennent à la fois le nombre réel et le nombre flottant. En séparant le terme flottant du terme d'erreur on peut donc obtenir un intervalle d'erreur $[\boldsymbol{e_a}]$ dont la largeur est souvent plus petite que la largeur de l'intervalle calculé par une arithmétique d'intervalle classique. De plus, il est possible d'utiliser des précisions différentes pour le terme d'erreur et le terme flottant. Si on veut suivre exactement les calculs effectués en machine, on utilisera la précision de la machine pour représenter $\boldsymbol{f_a}$, mais on peut utiliser plus de bits pour le calcul de $[\boldsymbol{e_a}]$ pour avoir un résultat encore plus précis.

Les calculs dans l'arithmétique de l'erreur globale se font comme suit. Un nombre réel x sera traduit en un nombre a à erreur globale tel que $\boldsymbol{f_a}$ est le nombre flottant le plus proche de x, noté $\uparrow_\sim (x)$, et l'erreur est la distance entre x et $\uparrow_\sim (x)$, notée $\downarrow_\sim (x)$. On a donc :

$$a = \uparrow_\sim (x)\vec{\epsilon_f} + \downarrow_\sim (x)\vec{\epsilon_e}\ .$$

Dans notre implémentation, la valeur $\downarrow_\sim (x)$, qui est censée être un nombre réel, sera encadrée par l'intervalle $[-u, u]$ où u est la moitié de la valeur du dernier bit dans la représentation de $\uparrow_\sim (x)$. Les opérations élémentaires sur les nombres à erreur globale sont données à la figure 6.9. Ainsi, l'addition de deux nombres à erreur globale a et b donne un nombre à erreur globale $c = \boldsymbol{f_c}\vec{\epsilon_f} + e_c\vec{\epsilon_e}$ tel que $\boldsymbol{f_c}$ la somme de $\boldsymbol{f_a}$ et $\boldsymbol{f_b}$ et e_c est la somme de e_a, e_b et l'erreur de représentation du résultat de $\boldsymbol{f_a} + \boldsymbol{f_b}$, soit $e_c = e_a + e_b + \downarrow_\sim (\boldsymbol{f_a} + \boldsymbol{f_b})$. Pour la soustraction, les règles sont similaires. Pour la multiplication, il y a un produit croisé de telle sorte que le terme d'erreur de b est multiplié par $\boldsymbol{f_a}$ et inversement. Un terme d'erreur du second ordre $e_a \cdot e_b$ est également introduit.

Ainsi, en utilisant l'arithmétique de l'erreur globale pour calculer $\boldsymbol{y_{n+1}}$ à partir de $\boldsymbol{y_n}$, on obtient non seulement la valeur flottante $\boldsymbol{y_{n+1}}$ mais également une surapproximation $[\boldsymbol{e_{n+1}}]$ de sa distance à la valeur réelle y^r_{n+1}. On a donc

$$\begin{aligned} y^r_{n+1} &\in \boldsymbol{y_{n+1}} + \left[e_{n+1}\right] \\ y^r_{n+1} - \boldsymbol{y_{n+1}} &\in \left[e_{n+1}\right]\ . \end{aligned} \tag{6.24}$$

$$a = \boldsymbol{f_a}\vec{\epsilon_f} + e_a\vec{\epsilon_e} \text{ and } b = \boldsymbol{f_b}\vec{\epsilon_f} + e_b\vec{\epsilon_e}$$

$$\begin{array}{rcl} a+b & = & \uparrow_\sim (\boldsymbol{f_a}+\boldsymbol{f_b})\vec{\epsilon_f} + (e_a+e_b+\downarrow_\sim (\boldsymbol{f_a}+\boldsymbol{f_b}))\vec{\epsilon_e} \\ a-b & = & \uparrow_\sim (\boldsymbol{f_a}-\boldsymbol{f_b})\vec{\epsilon_f} + (e_a-e_b+\downarrow_\sim (\boldsymbol{f_a}-\boldsymbol{f_b}))\vec{\epsilon_e} \\ a\times b & = & \uparrow_\sim (\boldsymbol{f_a}\times\boldsymbol{f_b})\vec{\epsilon_f} + \big(e_a\boldsymbol{f_b}+e_b\boldsymbol{f_a}+e_ae_b+\downarrow_\sim (\boldsymbol{f_a}\times\boldsymbol{f_b})\big)\vec{\epsilon_e} \end{array}$$

FIG. 6.9 – Règles de l'arithmétique de l'erreur globale.

Insistons encore sur l'intérêt d'utiliser une arithmétique d'erreur globale et non une arithmétique d'intervalle milieu/rayon par exemple. Pour cela, prenons l'exemple suivant [1] : on suppose que l'on dispose de nombres à virgule flottante avec une mantisse de 5 chiffres seulement. Un programme part de $y = 1$, et soustrait 10^6 fois la valeur 10^{-6} à y. Clairement, le résultat attendu (c'est-à-dire que l'on obtiendrait avec une arithmétique réelle) est 0. Un calcul en nombre flottant donne un résultat de 1, car chaque opération induit une élimination catastrophique (avec ce schémas de nombres flottants, on a $1-10^{-6}=1$). En utilisant une arithmétique d'intervalles reposant sur une représentation minimum/maximum, on obtient un intervalle d'une largeur de 10^{66}. Avec une représentation milieu/rayon, on obtient pour le milieu 1 et pour le rayon 1.0001 ; l'encadrement final est donc $[-0.0001, 2.0001]$, soit un intervalle de largeur 2. Avec l'arithmétique de l'erreur globale, on obtient un nombre flottant de 1, le terme d'erreur est $[-1.0001, -0.99977]$. L'encadrement finale est donc de largeur $4 \cdot 10^{-4}$. Cette différence s'explique par le fait que l'arithmétique de l'erreur globale est *dirigée* : le terme d'erreur n'est pas un rayon indiquant un cercle dans lequel se trouve le résultat réel, mais plutôt un vecteur pointant vers le nombre réel.

6.4.6 Pour résumer

En utilisant les formules (6.20), (6.22) et (6.24), on peut donc encadrer l'erreur au pas t_{n+1} en fonction de l'erreur au pas t_n :

$$\begin{array}{rcll} [\boldsymbol{y_{n+1}}] & = & \boldsymbol{y_{n+1}} + [\boldsymbol{\epsilon_{n+1}}] & (6.25) \\ \boldsymbol{y_{n+1}} & = & \Phi(y_n, h_{n+1}) & \\ [\boldsymbol{\epsilon_{n+1}}] & = & \dfrac{h^5}{120}\cdot\left(\dfrac{d^4 f}{dt^4}\big(\widetilde{[y_n]}\big) - \dfrac{d^5(\phi_n)}{dt^5}\big([t_n, t_{n+1}]\big)\right) + J\big(\psi_n, y_n + [\epsilon_n]\big)\cdot[\epsilon_n] + [\boldsymbol{e_{n+1}}]\,. & \end{array}$$

Nous nous intéressons maintenant au problème du choix du pas d'intégration h_{n+1} qui est un problème crucial pour la performance globale de l'intégration garantie.

6.4.7 Contrôle du pas d'intégration

L'équation (6.25) permet donc de calculer une borne sur l'erreur globale d'intégration. On s'intéresse maintenant au problème de rendre cette erreur la plus faible possible en choisissant un pas d'intégration optimal. On va donc fixer une certaine tolérance $tol > 0$ et essayer de maintenir ϵ_n inférieur à tol. Malheureusement, contrôler efficacement l'erreur globale pour la rendre inférieure à tol est une tâche très complexe [Sha05]. En effet, dans l'absolu, il faudrait faire deux intégrations : une première pour détecter les parties de l'espace où la fonction à intégrer est contractante et les parties où elle est expansive, puis adapter le pas d'intégration en fonction de ces zones pour limiter au maximum le wrapping effect. Cependant, même ainsi, on ne pourra pas forcément atteindre le seuil tol. En effet, dans l'équation (6.25), les termes χ_{n+1} et e_{n+1} sont très peu dépendant du pas d'intégration choisi et un contrôle du pas n'a donc que peu d'influences sur eux. Le terme χ_{n+1} dépend en effet nettement plus de la fonction à intégrer et nous choisissons

[1] Un programme C++ qui montre ces résultats est donné à l'adresse `http ://www.lix.polytechnique.fr/Labo/Olivier.Bouissou/progs/patriot.cc`

donc, dans un premier temps, de ne pas le prendre en compte pour contrôler le pas. Le terme e_{n+1} est essentiellement dépendant de la précision utilisé pour effectuer les calculs ; une solution pour le limiter est d'effectuer les calculs de $\boldsymbol{y_{n+1}}$ avec une librairie de nombres flottants multi-précision comme MPFR. Nous allons donc uniquement essayer de résoudre le problème suivant : étant donnée une tolérance relative *tol* définie par l'utilisateur, comment adapter la taille du pas à chaque étape pour que l'erreur de méthode sur un pas η_n soit inférieure à *tol*.

Le contrôle du pas d'intégration est un des facteurs clés dans la qualité d'un intégrateur numérique d'équations différentielles. En effet, les meilleures méthodes numériques calculent en parallèle de la solution approchée une estimation de l'erreur commise. Par exemple, la méthode de Runge-Kutta Cash-Karp effectue deux intégrations : une d'ordre 4 et une d'ordre 5. La différence entre les deux valeurs sert à estimer l'erreur commise [Sha05]. Le pas d'intégration suivant est alors choisi de telle sorte que l'erreur estimée soit maintenue sous une tolérance préalablement fixée par l'utilisateur. Le calcul de ce pas utilise des techniques assez similaires à celles que nous présentons par la suite (voir par exemple [Gus93]), à la différence notable que nous disposons non plus d'une estimation de l'erreur mais d'une surapproximation sûre.

Notre technique de contrôle du pas est une adaptation des méthodes de la théorie du contrôle [Gus91] pour le contrôle automatique de paramètres physiques. L'idée principale est que l'on voit l'erreur η_{n+1} comme une variable physique dépendant d'un paramètre d'ajustement h_n, le pas d'intégration. La valeur optimale pour η_{n+1} est *tol*, et le but de l'algorithme de modification du pas est d'amener η_{n+1} le plus près possible de *tol*. On sait que la dépendance entre η_{n+1} et h_n est de la forme $\eta_{n+1} = |\varphi_n| \cdot h_n^5$ où $|\varphi_n|$ est un terme proportionnel à la cinquième dérivée de la fonction ϕ_n calculée sur l'encadrement a priori de y_∞. Cela nous donne un premier algorithme de contrôle du pas appelé "contrôleur intégrale", qui fait l'hypothèse que $\varphi_{n+2} \approx \varphi_{n+1}$:

$$h_{n+1} = \left(\frac{tol}{|\varphi_{n+1}| \cdot h_n^5} \right)^{\frac{1}{5}} \cdot h_n \tag{6.26}$$

En effet, en suivant cette formule, on obtient

$$\eta_{n+2} = |\varphi_{n+2}| \cdot h_{n+1}^5 \approx |\varphi_{n+1}| \cdot \frac{tol}{|\varphi_{n+1}| \cdot h_n^5} \cdot h_n^5 = tol \ .$$

Cependant, l'hypothèse de base que $\varphi_{n+1} \approx \varphi_n$ n'est vraie que pour des problèmes triviaux, de sorte que cette méthode a tendance à sur-compenser les variations de l'erreur. On a donc des fortes variations de l'erreur ce qui conduit au final à de nombreux pas rejetés (quand on a $\eta_{n+1} > tol$). Pour contourner ce problème, on peut remplacer θ par $\theta \cdot tol$ dans l'équation (6.26), avec $\theta \in]0, 1[$, pour adoucir l'effet du contrôleur intégral. Une autre possibilité est de prendre en compte non pas juste un pas mais l'évolution de l'erreur sur les deux derniers pas pour estimer le pas suivant. Ainsi, si on se rend compte que l'erreur à tendance à grossir, on va compenser le contrôleur intégrale en réduisant plus fortement le pas. Au contraire, si on remarque que l'erreur tend à diminuer, on peut augmenter plus que prévu le pas. Pour cela, on va ajouter au contrôleur intégrale un terme proportionnel aux variations de l'erreur. Notons $r_n = |\varphi_{n+1}| \cdot h_n^5$. Le contrôleur proportionnel intégrale est alors donné par

$$h_{n+1} = \left(\frac{\theta.tol}{r_{n+1}} \right)^{k_i} \left(\frac{r_n}{r_{n+1}} \right)^{k_p} h_n, \tag{6.27}$$

où k_i et k_p sont des paramètres indiquant le poids respectif des termes intégrale ($\left(\frac{\theta.tol}{r_{n+1}} \right)^{k_i}$) et proportionnel ($\left(\frac{r_n}{r_{n+1}} \right)^{k_p}$). Idéalement, k_i et k_p doivent être choisis indépendemment pour chaque équation différentielle ; en pratique, nous choisissons les valeurs $k_1 = \frac{0.3}{5}$ and $k_p = \frac{0.2}{5}$ qui donnent de bons résultats.

6.5 Comparaison entre GRKLib et les autres méthodes

6.5.1 Comparaison expérimentale

Nous présentons maintenant des résultats expérimentaux permettant d'estimer la qualité de la méthode GRKLib par rapport aux méthodes existantes les plus reconnues, à savoir VNODE et AWA. Pour cela, nous choisissons 3 problèmes de Cauchy et mesurons, pour chacune des méthodes, la variation de la précision et du temps de calcul en fonction du temps final d'intégration. Nous effectuons ces mesures avec GRKLib, puis, pour chacune des méthodes VNODE et AWA, nous effectuons deux séries de mesures : une avec un ordre de 20 (c'est-à-dire que le développement en séries de Taylor est poussé jusqu'à l'ordre 20), et une avec un ordre de 5 (soit l'ordre de la méthode RK4). Nous présentons ci-dessous les 3 problèmes choisis (qui font partie des benchmarks classiques pour la comparaison d'algorithme d'intégration garantie) et les résultats expérimentaux obtenus, puis nous essaierons d'expliquer ces résultats.

Problème de contraction

Le premier problème que nous étudions est le problème de Cauchy suivant :

$$\dot{Y} = \begin{pmatrix} -0.4375 & 0.0625 & 0.2652 \\ 0.0625 & 0.4375 & 0.2652 \\ -0.2652 & 0.2652 & 0.375 \end{pmatrix} Y \quad Y_0 = \begin{pmatrix} 1.0 \\ 1.0 \\ 1.0 \end{pmatrix} . \tag{6.28}$$

C'est un problème dit de contraction car il peut s'écrire $\dot{y} = A.y$ où A est une matrice de contraction. Nous avons fait des comparaisons sur ce problème entre VNODE, AWA et GRKLib. Nous avons également étudié l'impact du paramètre "tolérance", défini par l'utilisateur, sur la précision du résultat. Nous avons ainsi mesuré la largeur de l'intervalle obtenu pour un temps d'intégration $T = 1000$ et pour des tolérances variant de 10^{-2} à 10^{-13}. Les résultats sont présentés à la figure 6.10.

Problème de rotation

Le second problème est le problème de Cauchy suivant :

$$\dot{Y} = \begin{pmatrix} 0 & -0.707107 & -0.5 \\ 0.707107 & 0 & 0.5 \\ 0.5 & -0.5 & 0 \end{pmatrix} Y \quad Y_0 = \begin{pmatrix} 1.0 \\ 1.0 \\ 1.0 \end{pmatrix} . \tag{6.29}$$

C'est un problème dit de rotation car il peut s'écrire comme $\dot{Y} = A \cdot y$ où A est une matrice de rotation. Nous avons également étudié, pour ce problème, l'effet de la dimension du problème de Cauchy sur les performances de l'intégration. Ainsi, nous avons augmenté la matrice A décrite ci-dessus pour des dimensions allant de 3 à 140 et mesuré à chaque fois le temps nécessaire pour effectuer un pas de calcul. Les résultats sont présentés à la figure 6.11.

Problème de Lorenz

Nous avons également étudié le comportement de GRKLib sur un problème fortement non linéaire, connu pour être très instable. Il s'agit des équations de Lorenz qui décrivent les phénomènes de convection dans un fluide idéal en deux dimensions [Lor63].

$$\begin{cases} \dot{y_1} = \sigma.(y_2 - y_1) \\ \dot{y_2} = y_1.(\rho - y_3) - y_2 \\ \dot{y_3} = y_1.y_2 - \beta.y_3 \end{cases} \quad \begin{cases} y_1(0) = 15.0 \\ y_2(0) = 15.0 \\ y_3(0) = 36.0 \end{cases} \tag{6.30}$$

Les paramètres ρ, σ et β sont fixes et nous avons choisi les valeurs suivantes : $\rho = 28$, $\sigma = 10$ et $\beta = 8/3$. Nous avons étudié l'impact de l'erreur initiale sur la stabilité de l'intégration garantie. Ainsi, nous avons pris le même problème mais en ajoutant une erreur initiale allant de 10^{-9} à 10^{-4}, et nous avons mesuré le temps final que l'on pouvait alors atteindre. Les résultats sont donnés à la figure 6.12.

Discussion

D'un point de vue efficacité des calculs, GRKLib se situe à mi-chemin entre VNODE d'ordre 20 et VNODE d'ordre 5 (que nous noterons dans la suite VNODE(5)) : en moyenne, GRKLib est 5 fois plus rapide que VNODE(5) et 5 fois plus lent que VNODE. Dans tous les exemples, AWA est plus lent que GRKLib, la différence étant grande si on limite AWA à l'ordre 5. Cette différence de rapidité s'explique essentiellement par le fait que GRKLib nécessite un nombre de pas bien plus grand que VNODE pour obtenir une précision similaire. En effet, l'ordre relativement faible de GRKLib impose des pas petits si on veut une erreur locale faible. Ainsi, si le temps de calcul par pas d'intégration est bien plus faible pour GRKLib que pour VNODE, ce dernier est plus performant d'un point de vue global car il exécute beaucoup moins de pas : en moyenne sur les tests effectués, VNODE effectue entre 12 et 13 fois moins de pas que GRKLib pour atteindre le temps final. En comparaison à VNODE(5), l'efficacité de GRKLib est meilleure grâce à un choix d'implémentation différent : les dérivées nécessaires au calcul sont générées par calcul formel, avant la compilation, il n'est donc pas nécessaire de les recalculer à chaque pas. Le temps de calcul par pas est donc plus faible pour GRKLib que pour VNODE(5), et le nombre de pas effectués est en moyenne 40 fois plus élevé pour VNODE(5).[2] Ceci explique la différence d'efficacité globale.

L'ordre faible de la méthode GRKLib a également un impact sur la précision des calculs effectués. On constate en effet dans les 3 cas que VNODE est plus précis que GRKLib, dont l'erreur est du même ordre de grandeur que pour VNODE(5). Si on regarde les courbes correspondant à la précision pour chaque exemple (courbes du milieu dans les figures 6.10 à 6.12), on voit que la différence entre VNODE et GRKLib est constante quelque soit la durée totale de l'intégration. La stabilité des méthodes VNODE, VNODE(5) et GRKLib est donc tout à fait comparable. La différence dans les résultats obtenus par GRKLib et VNODE s'explique en fait par le fait que, puisque VNODE est d'ordre 20, il peut à moindre coût choisir des pas d'intégration tel que l'erreur introduite à chaque pas est largement inférieure à la tolérance fixée par l'utilisateur. Ainsi, l'erreur introduite par les premiers pas est très faible. Au contraire, pour GRKLib ou VNODE(5), la largeur des pas est choisie de telle sorte que l'erreur locale est proche de la tolérance. Ainsi, dès le départ l'erreur est plus élevée que pour VNODE. Ensuite, l'erreur locale introduite à chaque pas est souvent négligeable par rapport à l'erreur propagée, de telle sorte que c'est en fait la stabilité de la méthode qui joue plus que son ordre. Remarquons cependant que, si on utilise GRKLib avec un pas fixe pour l'équation (6.29), on peut obtenir des résultats aussi bon qu'avec VNODE, pour un temps de calcul élevé (voir figure 6.11, en bas).

Comme on peut s'y attendre, la précision des résultats est largement dépendante de la tolérance fixée (figure 6.10, en bas). Remarquons cependant qu'une fois passé un seuil (environ 10^{-11} pour l'exemple de la figure 6.10), on ne peut plus augmenter la précision ; pour une tolérance très faible, la taille de chaque pas est si faible que les erreurs de calcul dues à l'utilisation de nombres à virgule flottante deviennent prépondérantes sur l'erreur due à la méthode d'intégration.
Enfin, la qualité des résultats dépend fortement de l'erreur initiale, surtout pour les problèmes les plus instables. Ainsi, pour le problème de Lorenz (figure 6.12, en bas), on voit bien que plus l'erreur initiale est élevée, plus vite la largeur de l'intervalle grossit, si bien que la méthode (que ce soit VNODE ou GRKLib) diverge rapidement.

Notre approche qui consiste à transformer une méthode numérique classique en une méthode garantie apporte plusieurs avantages indéniable. Tout d'abord, la méthode est très générale et peut être appliquée à bien d'autres algorithmes numériques (que ce soit un algorithme d'intégration d'équations différentielles ou pas). Ainsi, nous l'avons appliqué avec succés pour obtenir un algorithme garanti de calcul d'intégrale par la méthode de Simpson. De plus, l'algorithme ainsi obtenu calcule non seulement un encadrement du résultat réel (dans notre cas, la solution de l'équation différentielle), mais aussi la valeur flottante fournie par l'algorithme numérique sous-jacent. Cette information supplémentaire permet donc de mesurer la qualité de l'algorithme numérique ce qui aide à convaincre les numériciens de choisir une méthode ou une autre. Enfin, les choix d'implémentations que nous avons effectués apportent un plus par rapport aux méthodes exis-

[2] La différence est surtout notable pour les problèmes non-linéaires : sur le problème de Lorenz, GRKLib fait en moyenne 100 fois moins de pas que VNODE(5).

tantes. En effet, comme nous n'avons besoin que de la dérivée cinquième de ψ et de la dérivée quatrième de f pour calculer le terme d'erreur (voir l'équation (6.20)), il est très coûtex d'utiliser la différentiation automatique pour les obtenir : on calculer pour rien les premières dérivées. Nous avons donc choisi de générer, au moment de la compilation, du code C++ qui calcule directement les dérivées nécessaires, ce qui permet un gain important du temps de calcul par pas d'intégration. En manipulant de manière formelle les dérivées nécessaires, on peut donc, par le jeu des optimisations du compilateur, améliorer de manière conséquente l'efficacité des calculs.

6.5.2 Comparaison formelle

Nous essayons ici d'effectuer une comparaison formelle entre l'algorithme d'intégration garantie utilisé par GRKLib et celui utilisé par les techniques à base de séries de Taylor décrit au chapitre 3. Rappelons les formules permettant de calculer l'encadrement de la solution au temps t_{n+1} en fonction de l'encadrement au temps t_n pour les méthodes à base de série de Taylor (équation (6.31)) et pour GRKLib (équation (6.32)).

$$\begin{aligned}[\boldsymbol{y_{n+1}}]^{st} &= \widehat{\boldsymbol{y_n}} + \sum_{i=1}^{N-1} h_{n+1}^i F^{[i]}(\widehat{y_n}) + \\ &\quad \left(I + \sum_{i=1}^{N-1} h_{n+1}^i J\big(F^{[i]}, [\boldsymbol{y_n}]\big)\right)\big([\boldsymbol{r_n}]\big) + h_{n+1}^N F^{[N]}([\widetilde{\boldsymbol{y_n}}])\end{aligned} \tag{6.31}$$

$$\begin{aligned}[\boldsymbol{y_{n+1}}]^{g} &= \boldsymbol{y_{n+1}} + [\boldsymbol{\epsilon_{n+1}}] \\ [\boldsymbol{\epsilon_{n+1}}] &= J\big(\psi_n, y_n + [\epsilon_n]\big) \cdot [\epsilon_n] + \frac{d^4F}{dt^4}\big([\widetilde{y_n}]\big) - \frac{d^5(\phi_n)}{dt^5}\big([t_n, t_{n+1}]\big) + [\boldsymbol{e_{n+1}}] \\ \boldsymbol{y_{n+1}} &= \Phi\big(\boldsymbol{y_n}, h_n\big)\end{aligned} \tag{6.32}$$

Dans les formules (6.31) et (6.32), $[\boldsymbol{y_{n+1}}]^{st}$ représente l'encadrement obtenu par la méthode des séries de Taylor et $[\boldsymbol{y_{n+1}}]^{g}$ par la méthode GRKLib. Les formules donnant chacun de ces termes sont très semblables, même si elles ont été obtenues par des raisonnements très différents. Dans chaque cas, l'encadrement à t_{n+1} est obtenu comme la somme d'un nombre flottant (le point d'approximation) et d'un terme d'erreur. Ce terme d'erreur est lui-même calculé comme la somme d'un terme représentant l'erreur introduite par le pas $[t_n, t_{n+1}]$ et d'un terme représentant la propagation de l'erreur présente à t_n. Comparons deux-à-deux chacun de ces termes.

Point d'approximation Pour la méthode des série de Taylor d'ordre N, le point d'approximation à $t = t_{n+1}$, $\boldsymbol{y_{n+1}}$, est calculé par

$$\boldsymbol{y_{n+1}}^{st} = \widehat{\boldsymbol{y_n}} + \sum_{i=1}^{N} h_{n+1}^i F^{[i]}(\widehat{y_n}) \tag{6.33}$$

alors que pour GRKLib, on a

$$\boldsymbol{y_{n+1}}^{g} = \boldsymbol{y_n} + \frac{h_n}{6}(k_1 + 2k_2 + 2k_3 + k_4) \tag{6.34}$$

où les termes k_i sont donnés par l'équation (6.16). Le calcul de l'équation (6.34) nécessite 4 évaluations de F, alors que le calcul de l'équation (6.33) demande l'évaluation de F ainsi que de ses N premières dérivées en $\widehat{\boldsymbol{y_n}}$. D'un point de vue efficacité des calculs, le calcul des dérivées successives par différentiation automatique est clairement plus coûteux que l'évaluation de la fonction F. Moore a montré que si la fonction F est formée de N opérations élémentaires (addition, soustraction, multiplication), alors on peut générer les K premiers coefficients de Taylor en un temps $N * K^2$ [Moo66]. On a donc un net avantage à utiliser une méthode numérique ne reposant

pas sur le développement en série de Taylor pour calculer le point d'approximation, d'autant plus qu'on bénéficie des optimisations du compilateur qui rendent l'évaluation de F plus rapide, alors que les coefficients de Taylor sont calculés par différentiation automatique à l'exécution ce qui limite les capacités d'optimisation de compilation.

Du point de vue de la qualité du point obtenu, les deux formules donnent des points très proches si on utilise $N = 5$ pour la méthode par séries de Taylor.

Erreur sur un pas L'erreur introduite lors du pas $[t_n, t_{n+1}]$ pour la méthode des séries de Taylor d'ordre N est donnée par

$$\eta_{n+1}^{st} = h_{n+1}^{N} F^{[N]}([\widetilde{\boldsymbol{y_n}}]) \tag{6.35}$$

alors que pour GRKLib, on a (si on oublie momentanément l'erreur de calcul)

$$\eta_{n+1}^{g} = \frac{d^4 F}{dt^4}\big([\widetilde{y_n}]\big) - \frac{d^5(\phi_n)}{dt^5}\big([t_n, t_{n+1}]\big) \ . \tag{6.36}$$

Remarquons que ϕ_n s'exprime comme la somme de quatre évaluations de F, de sorte que $\frac{d^5(\phi_n)}{dt^5}$ est proportionnel à $\frac{d^5 F}{dt^5}$. De même, dans le terme η_{n+1}^{st}, $F^{[N]}$ est proportionnel à $\frac{d^N F}{dt^N}$. On voit donc que, si $N = 5$, les erreurs sur un pas sont tout à fait comparable. La différence vient surtout de la méthode de calcul de $\widetilde{y_n}$. Pour GRKLib, nous avons considéré que le terme η_{n+1} n'était pas le plus important dans l'erreur globale, et donc on utilise un encadrement a priori assez large obtenu par la méthode dite de l'encadrement constant. Les outils à base de séries de Taylor utilisent maintenant des techniques dites d'ordre supérieur pour calculer cet encadrement, ce qui permet d'avoir un pas plus élevé et un encadrement plus fin.

D'un point de vue de l'efficacité des calculs, l'implémentation de GRKLib est plus performante car la fonction $\frac{d^5(\phi_n)}{dt^5}$ est calculée lors de la compilation puis compilée. On a donc la encore un net avantage par rapport aux méthodes utilisant la différentiation automatique.

Propagation de l'erreur Pour les méthodes à base de séries de Taylor, l'erreur au pas n est propagé dans l'erreur au pas $n + 1$ par :

$$\chi_{n+1}^{st} = \left(I + \sum_{i=1}^{N-1} h_{n+1}^{i} J\big(F^{[i]}, [\boldsymbol{y_n}]\big) \right) \big([\boldsymbol{r_n}]\big) \tag{6.37}$$

alors que pour GRKLib, cette propagation s'exprime par :

$$\chi_{n+1}^{g} = J\big(\psi_n, y_n + [\epsilon_n]\big) \cdot [\epsilon_n] \ . \tag{6.38}$$

La fonction ψ_n intervenant dans l'équation (6.38) peut s'exprimer en composant 4 fois F avec elle même (équation (6.18)). Les règles de calcul de la jacobienne montre alors que le terme χ_{n+1} peut s'exprimer en fonction des termes $J(F^{[i]}, y_n + [\epsilon_n])$, ce qui montre que les deux termes χ_{n+1}^{st} et χ_{n+1}^{g} sont très proches. Le terme χ_{n+1}^{st} est croissant avec l'ordre N car on ne fait qu'ajouter des termes à la somme dans l'équation (6.37). Le plus important dans le calcul de χ_{n+1} est la méthode utilisée pour réduire le wrapping effect. Pour GRKLib comme pour les outils les plus performants, on utilise la méthode de factorisation QR de Löhner (voir chapitre 3) qui donne de très bon résultats. Ceci explique les résultats expérimentaux qui montrent une stabilité équivalent pour GRKLib et VNODE, que ce soit à l'ordre 20 ou à l'ordre 5.

Conclusion On voit donc que d'un point de vue théorique, les méthodes d'intégration garantie à base de séries de Taylor et la méthode GRKLib sont très proches ; les formules permettant de calculer l'erreur globale sont très comparables. Le gros avantage de GRKLib est l'utilisation de l'arithmétique de l'erreur globale à la place d'une arithmétique d'intervalle pour calculer $\boldsymbol{y_{n+1}}$. Comme nous l'avons vu en section 6.4.5, le domaine de l'erreur globale est plus stable que les calculs à base d'intervalle, même si on utilise les fonctions d'inclusion centrées. Par ailleurs, GRKLib

apporte plus d'informations que les méthodes à base de séries de Taylor : on ne donne pas uniquement un encadrement de la solution réelle mais on indique également la qualité de la résolution numérique par la méthode de RK4. Remarquons que les techniques que nous avons présenté dans ce chapitre pour transformer la méthode RK4 numérique en une méthode d'intégration garantie peuvent facilement se généraliser à n'importe quelle méthode numérique [3]. Ainsi, nous voulons intégrer dans l'outil GRKLib d'autres méthodes numériques (par exemple, les méthodes d'Euler, de Heun, des méthodes de Runge-Kutta d'ordre supérieur, voir des méthodes à base de séries de Taylor) pour avoir à notre disposition un ensemble d'intégrateurs garantis. On pourra alors choisir le plus adapté au problème de Cauchy que l'on veut intégrer.

Pour l'implémentation de GRKLib, nous avons choisi d'utiliser au maximum les templates C++ et la différentiation symbolique pour accélérer le temps de calcul. Cela s'est révélé concluant : grâce à un pré-traitement, on transfert une partie du temps de calcul de l'exécution à la compilation, et on obtient ainsi un temps d'exécution par pas d'intégration bien plus faible que pour VNODE (d'ordre 5 ou 20). Si actuellement le calcul des dérivées ne se fait pas intégralement lors de la compilation (il faut utiliser un programme externe), la prochaine version de GRKLib, écrite en Objective Caml et utilisant CAMLP4, sera plus automatique. Remarquons cependant qu'il reste des problèmes à résoudre pour rendre GRKLib vraiment compétitive face à VNODE. Tout d'abord, notre méthode de contrôle du pas nous semble trop peu précise : la taille du pas d'intégration a tendance à fortement varier lors des calculs ce qui augmente la surapproximation de l'erreur. Un contrôle du pas plus souple et plus subtil permettrait de stabiliser le pas à une taille optimale. Par ailleurs, la taille du pas est actuellement faible car la méthode RK4 n'est que d'ordre 5. L'intégration d'autres méthodes numériques d'ordre plus élevé à GRKLib permettrait de choisir dynamiquement à la fois le pas et la méthode d'intégration la plus adaptée, pour avoir ainsi un contrôle dynamique de l'ordre d'intégration.

[3] Nous avons par exemple utilisé ces idées pour implémenter une version garantie de la méthode de calcul d'intégrale de Simpson.

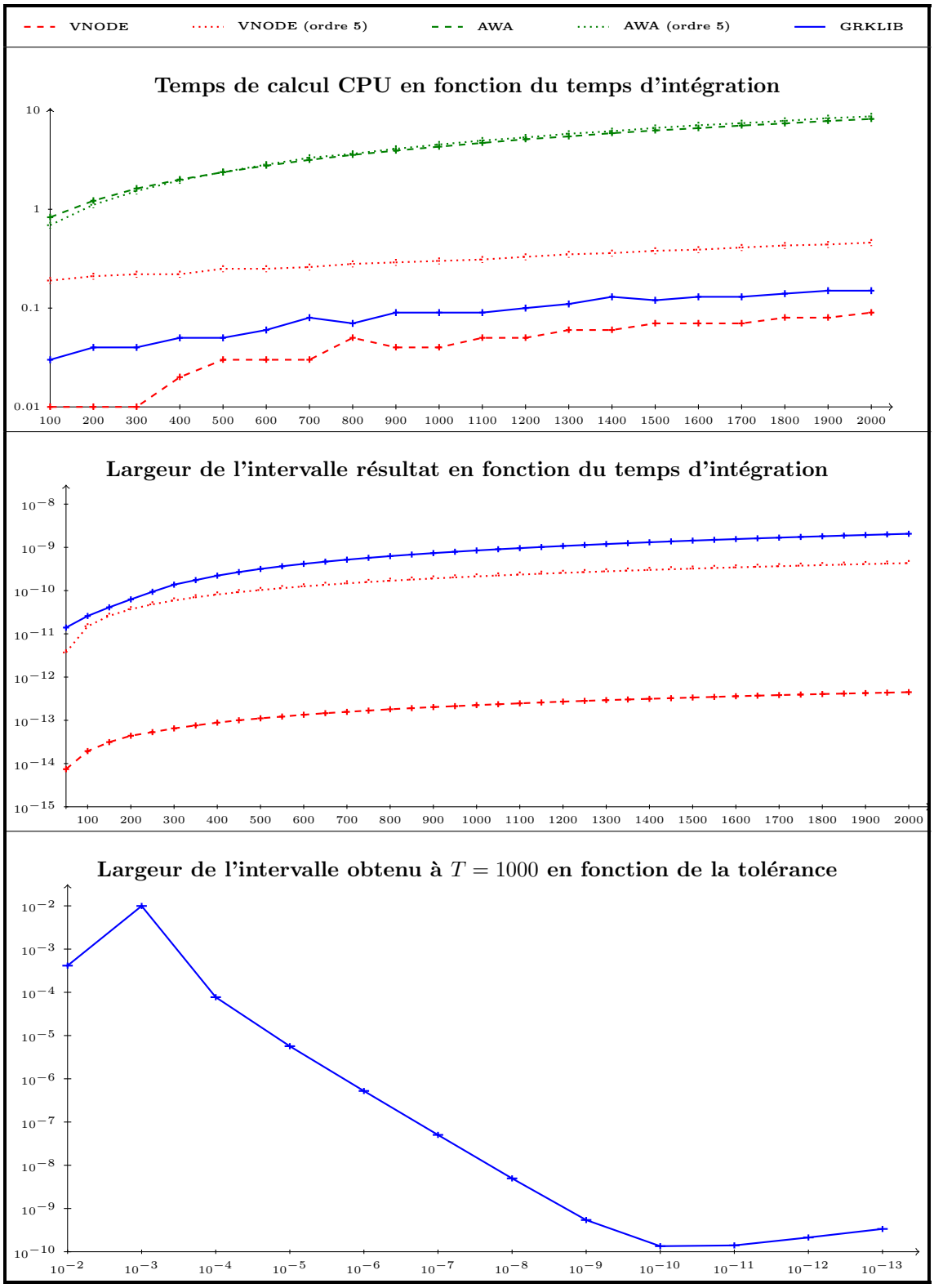


FIG. 6.10 – Résultats obtenus pour le problème de contraction.

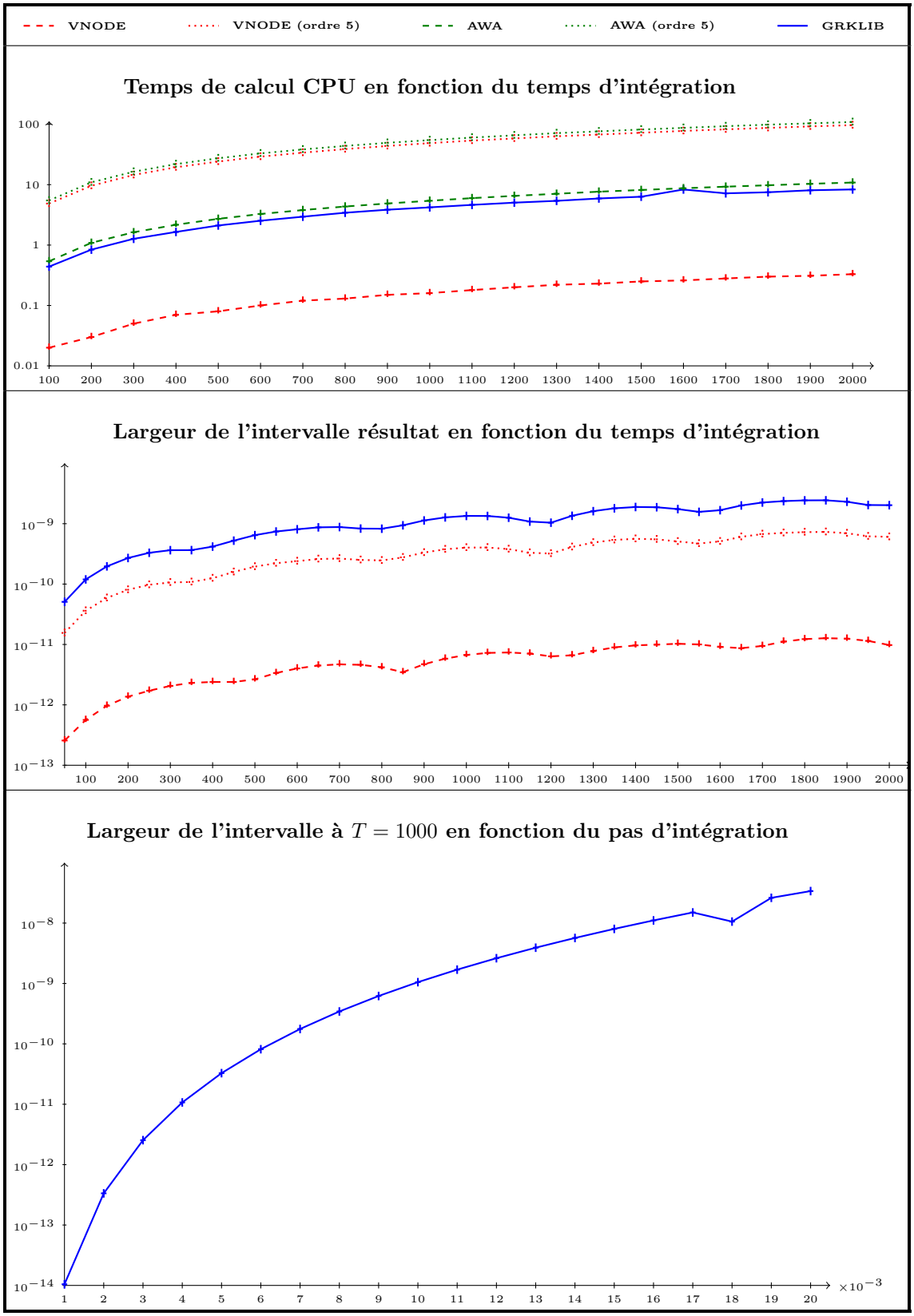


FIG. 6.11 – Résultats obtenus pour le problème de rotation.

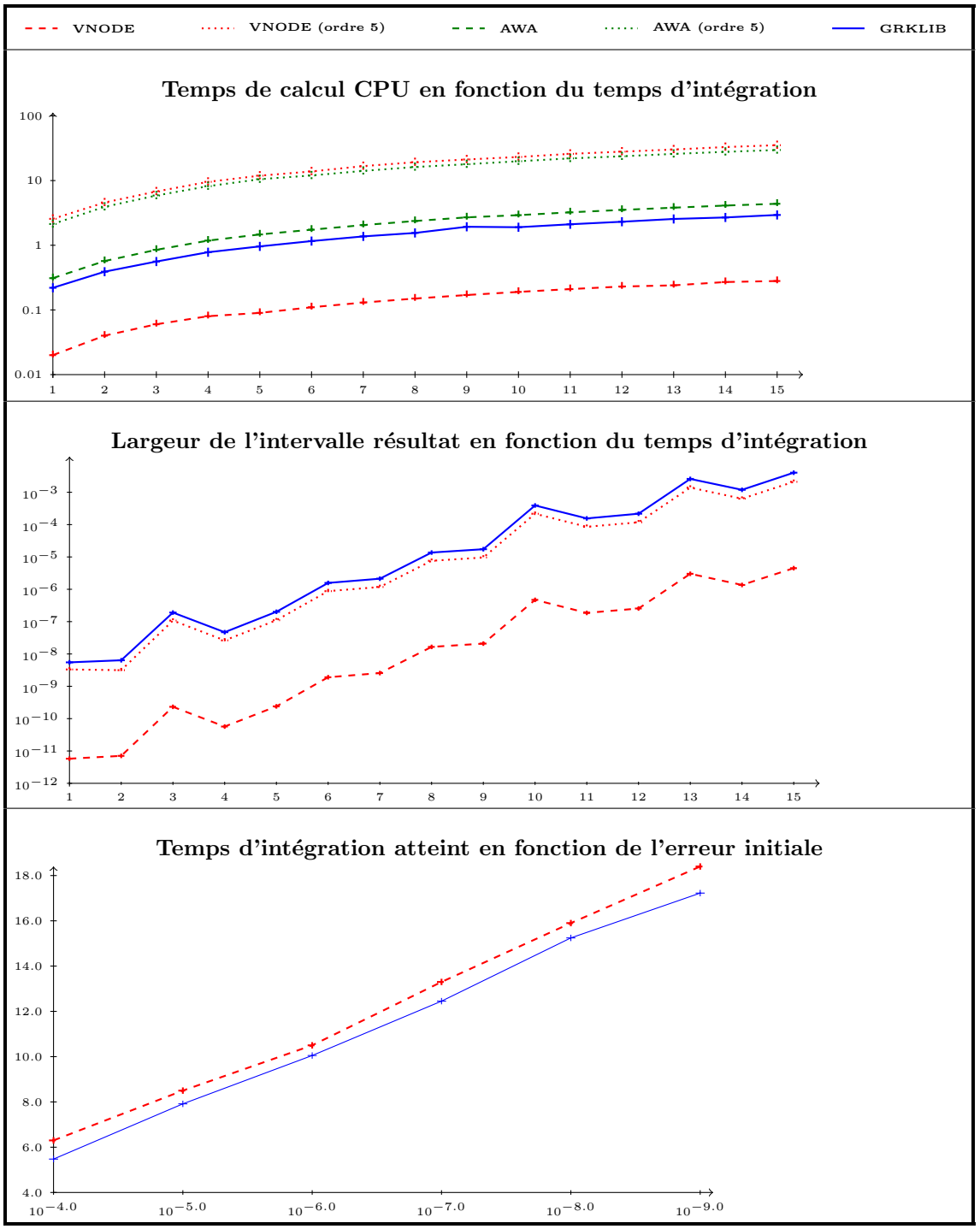


FIG. 6.12 – Résultats obtenus pour le problème de Lorenz.

CHAPITRE 7

Abstraction globale

Dans ce chapitre nous montrons comment on peut utiliser les abstractions définies au chapitre 6 pour effectuer une analyse du système hybride dans son ensemble. La principale difficulté provient de ce que nous ne connaissons pas les actions du programme sur l'environnement continu, c'est-à-dire que nous ne savons pas à quels instants la dynamique continue est changée par un actionneur. Nous allons donc dans un premier temps analyser le programme seul pour construire une abstraction de l'impact qu'a le programme sur son environnement, puis nous calculerons une surapproximation des trajectoires continues en utilisant cette abstraction du programme et l'abstraction définie au chapitre 6. Cette technique ne sera cependant utilisable que pour une classe bien définie de programmes. Nous commençons donc par présenter intuitivement notre technique d'analyse du système hybride et nous définissons les types de programmes pour lesquels cette technique s'applique (section 7.1), puis nous présentons une abstraction des fonctions booléennes par un graphe de décision (section 7.2). Ensuite, nous expliquons comment utiliser ces graphes pour obtenir une représentation abstraite de l'action du programme sur son environnement (section 7.3), et enfin nous expliquons comment utiliser ces techniques pour analyser le système hybride dans son intégralité.

7.1 Principe de l'analyse du système hybride

Comme nous l'avons vu, la principale difficulté dans la définition de la sémantique du système hybride est la prise en compte des actionneurs qui modifient la dynamique continue. L'analyse de la partie continue présentée au chapitre 6 oublie ces actionneurs et propose une abstraction de l'évolution continue entre deux instants auxquels les actionneurs ont été activés. Pour obtenir une abstraction de l'évolution du système hybride, nous devons donc prendre en compte des changements dans la dynamique continue. Les méthodes d'analyse pour les modèles classiques de systèmes hybrides (automates hybrides, calcul de processus hybride) font généralement des hypothèses simplificatrices sur l'évolution continue permettant de détecter facilement les changements de mode. Nous avons choisi au contraire d'autoriser n'importe quelle évolution continue décrite sous la forme d'une équation différentielle (potentiellement non linéaire). Pour prendre en compte les actionneurs dans l'abstraction de l'évolution du système hybride, nous allons effectuer une pré-analyse du programme qui fournira des critères spatiaux permettant de détecter les instants où la dynamique continue change.

L'idée de base sur laquelle se repose cette pré-analyse est la suivante : pour construire une abstraction de l'évolution continue, il faut connaître la fonction F de la sémantique continue (voir chapitre 5, section 5.5), c'est-à-dire qu'il faut avoir un partitionnement de $\mathbb{R}_+$ tel que, sur chaque

partie, la fonction F est égale à une des dynamiques du modèle hybride. Dans certains cas, on peut transformer ce partitionnement du temps en un partitionnement de l'espace, c'est-à-dire transformer les contraintes du type "à partir de l'instant $t = t_i$, la dynamique est dictée par la fonction F_k" par une contrainte du type "si $y \in R_i$, alors c'est F_k qui donne la dynamique". En effet, les changements de dynamique sont dûs à l'exécution d'une instruction ***act***. L'exécution ou non de cette instruction dépend essentiellement de l'ensemble des valeurs reçues par les capteurs, c'est-à-dire de la suite des entrées du programme. Si on suppose que l'exécution de cette instruction dépend *uniquement* de la *dernière* valeur reçue, alors on pourra construire, à partir du programme, un ensemble de régions telles que, si le programme reçoit comme entrée une valeur de cette région, alors l'actionneur sera activé. Plus précisément, nous ferons l'hypothèse que le programme P est invariant dans le temps (voir définition 7.1).

Définition 7.1 (Programme invariant dans le temps) Nous dirons qu'un programme P de **H-SIMPLE** est *invariant dans le temps* si il vérifie l'hypothèse suivante : si à un instant t, la dernière valeur fournie par un capteur est X et l'instruction $\boldsymbol{act}.m!k$ a été exécutée, alors pour tout instant $t' \neq t$, si les capteurs fournissent de nouveau la valeur X, l'instruction $\boldsymbol{act}.m!k$ sera encore exécutée.

Cette définition suppose donc un déterminisme temporel des actions du programme P sur l'environnement : la prise de décision (l'instruction $\boldsymbol{act}.m!k$ est-elle exécutée ?) ne doit dépendre que de la dernière valeur reçue et pas de tout l'historique d'exécution du programme.

Remarque Cette hypothèse nous empêche de considérer des programmes utilisant une corrélation entre les valeurs successives fournies par le capteur. Ainsi, on ne pourra pas traiter un programme filtrant ses entrées grâce à un filtre linéaire d'ordre n supérieur à 2 car les décisions que prend le programme dépendent des n dernières entrées. Cette hypothèse limitatrice peut sûrement être levée, mais l'abstraction de l'effet du programme sur son environnement est alors nettement plus compliquée à calculer (voir le chapitre de conclusion pour quelques pistes dans cette direction).

Exemple 7.1 Le programme

$$P ::= \boldsymbol{while}\ 1\ \boldsymbol{do}(sens.y?X;\ \boldsymbol{if}\ y > 1\ \boldsymbol{then}\ act.0!0;\ \boldsymbol{else}\ act.0!1;)$$

est invariant dans le temps. En revanche, le programme

$$P' ::= \boldsymbol{while}\ 1\ \boldsymbol{do}(sens.y?X;\ z = z + y;\ \boldsymbol{if}\ z > 1\ \boldsymbol{then}\ act.0!0;\ \boldsymbol{else}\ act.0!1;)$$

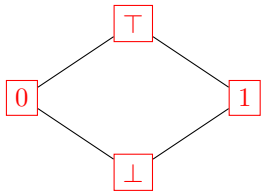
ne l'est pas car l'exécution d'un actionneur dépend de toutes les valeurs reçues depuis le début du programme.

Si on suppose que les programmes sont invariants dans le temps, on peut alors voir l'action d'un programme sur son environnement continu comme un ensemble de fonctions de la forme "étant donné une entrée X, l'instruction $\boldsymbol{act}.m!k$ est-elle exécutée ?" pour chaque instruction $\boldsymbol{act}.m!k$ du programme. Autrement dit, l'effet du programme sur l'environnement peut être abstrait par un ensemble fini de fonctions booléennes. Si on arrive à construire chacune de ces fonctions, on pourra alors construire une abstraction complète de l'évolution continue : il suffit, à chaque pas de calcul, de tester chaque fonction booléenne ; si une d'entre elles vaut 1 (ou *vrai*), alors on choisira la dynamique imposée par l'actionneur correspondant pour le pas de calcul suivant. Cependant, la construction de ces fonctions n'est pas possible en général : le problème de savoir si une instruction d'un programme va être exécutée est un problème indécidable, et nous allons donc construire une abstraction, sous forme d'un arbre régulier, des fonctions booléennes.

7.2 Abstraction des fonctions booléennes

Dans cette section, nous définissons une algorithme d'abstraction des fonctions réelles à valeur dans $\mathbb{B} = \{0, 1\}$. Cet algorithme utilise des techniques classiques d'analyse par intervalles pour construire une représentation abstraite des fonctions booléennes. Dans toute cette section, $\mathbb{B}_\perp$ représentera le treillis complet des valeurs booléennes donné par la définition 7.2 ci-dessous.

Définition 7.2 (Treillis complet des valeurs booléennes) Nous noterons $\mathbb{B}_\perp$ le treillis complet des valeurs booléennes tel que $\mathbb{B}_\perp = \{\perp, 0, 1, \top\}$ muni de l'ordre $\sqsubseteq$ représenté par le diagramme de Hasse ci-dessous. L'élément 0 représente la valeur "faux", 1 représente la valeur "vrai".



7.2.1 Représentation d'une fonction booléenne

Nous allons abstraire une fonction booléenne par sa représentation spatiale, c'est-à-dire un recouvrement de son domaine de définition en un ensemble de rectangles tel que la fonction est constante sur chaque rectangle. Cette représentation est donnée sous forme d'arbres réguliers appelés *octree*.

Définition 7.3 (Recouvrement d'un hyperrectangle) Soit $n \in \mathbb{N}$ et $b \in I_{\mathbb{R}}^n$ un hyperrectangle de dimension n. On dira qu'un ensemble $I(b) = \{b_1, \ldots, b_m\}$ d'éléments de $I_{\mathbb{R}}^n$ est un recouvrement de b si et seulement si $b = \cup_{i=1}^m b_i$.

Nous allons utiliser des représentations *régulières* des fonctions booléennes, c'est-à-dire que le recouvrement du domaine de définition sera obtenu en divisant les intervalles de définition dans chaque direction en deux. Cette technique n'est pas forcément optimale dans tous les cas, mais elle est la plus générale et la plus simple à mettre en place lorsque l'on considère des fonctions booléennes incluant des calculs potentiellement non linéaires.

Définition 7.4 (Recouvrement régulier d'un hyperrectangle) Soit $b \in I_{\mathbb{R}}^n$ un rectangle de dimension n. On notera $\mathit{Split}(b)$ le recouvrement de b tel que $|\mathit{Split}(b)| = 2^n$, et si $b = (b_1,\ b_2, \ldots,\ b_n)$ avec $\forall i \in [1, n],\ b_i \in I_{\mathbb{R}}$, alors on a :

$$\mathit{Split}(b) = \left\{(b'_1, \ldots, b'_n)\ :\ \forall i \in [1, n],\ b'_i = [\underline{b_i}, mid(b_i)] \text{ ou } b'_i = [mid(b_i), \overline{b_i}]\right\}\ .$$

Rappelons que pour un intervalle $x \in I_{\mathbb{R}}$, $\underline{x}$ est la borne inférieure de x, $\overline{x}$ sa borne supérieure et $mid(x) = \frac{\overline{x}-\underline{x}}{2}$ est le milieu de x.

Exemple 7.2 En dimension 2, on a par exemple

$$\mathit{Split}\big(([0, 10], [-2, 0])\big) = \big\{\big(([0, 5], [-2, -1])\big), \big(([0, 5], [-1, 0])\big), \big(([5, 10], [-2, -1])\big), \big(([5, 10], [-1, 0])\big)\big\}\ .$$

Définition 7.5 (Octrees) Un *octree* de dimension n est un graphe dirigé, acyclique, ayant une racine unique et deux types de noeuds :
- les noeuds non terminaux de la forme $N(b)$, avec $b \in I^n_{\mathbb{R}}$;
- les noeud terminaux de la forme $V(v)$ avec $v \in \{0, 1, \top\}$;

et tel que pour tout $b \in I^n_{\mathbb{R}}$, le noeud $N(b)$ a soit un noeud fils de la forme $V(v)$ soit 2^n noeuds fils non terminaux $N_1(b_1),\ N_2(b_2), \ldots,\ N_{2^n}(b_{2^n})$ avec $\{b_i\ :\ i \in [1, 2^n]\} = \mathit{Split}(b)$. Le noeud racine d'un octree T doit être un noeud non terminal, et on dira que T est de domaine $b \in I^n_{\mathbb{R}}$ si son noeud racine est $N(b)$. On notera $b = \mathit{dom}(T)$.

Remarque (1) Le terme *octree* est généralement utilisé pour le cas de la dimension 3 uniquement ; en dimension 2, on parle de *quadtree* Par abus de notation, nous utilisons le même terme pour toutes les dimensions.

Remarque (2) Une autre représentation possible des fonctions booléennes consiste à utiliser des Diagrammes de Décision Intervalles (*Interval Decision Diagrams*, ou IDD [Str99]) qui sont une généralisation des *Binary Decision Diagrams*, ou BDD. Cette forme d'arbre permet généralement une représentation plus compacte des fonctions booléennes car elle autorise un partitionnement non régulier du domaine de définition. Cependant, elle est moins pratique à utiliser pour la représentation de fonctions fortement non linéaires, ce qui sera souvent le type de fonctions que nous rencontrerons. En effet, les fonctions booléennes qui nous intéressent sont définies par la partie de contrôle des programmes embarqués qui contiennent souvent des calculs non linéaires (par exemple pour des filtres) ainsi que des instructions de branchement conditionnel. L'algorithme de construction de la représentation que nous utiliserons (voir section 7.2.2) est un algorithme de *branch and bound*, et nous devons donc pouvoir facilement calculer un partitionnement du domaine de la fonction. La représentation par octree nous semble donc plus adaptée.

Exemple 7.3 La figure 7.1 représente un octree de dimension 2 et de profondeur 3. Les noeuds non terminaux sont représentés par des rectangles et les noeuds terminaux par des cercles.

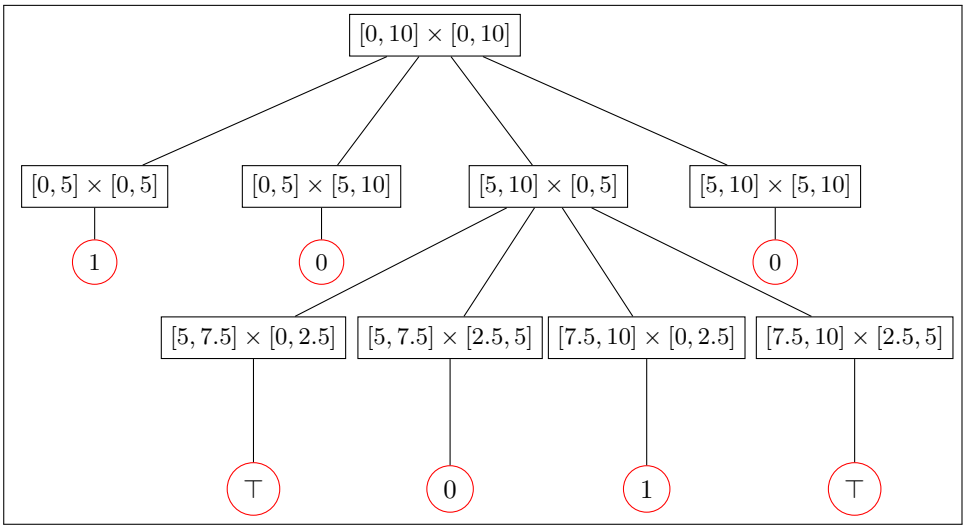


FIG. 7.1 – Exemple d'un octree de dimension 2 et de profondeur 3.

Intuitivement, un octree T est la représentation de toutes les fonctions booléennes f telles que, si T possède un noeud $N(b)$ avec un seul noeud fils $V(v)$, avec $b \in I^n_{\mathbb{R}}$ et $v \in \mathbb{B}_\perp$, alors f vérifie $f(x) \sqsubseteq v$ pour tout $x \in b$. L'ordre $\sqsubseteq$ est ici l'ordre sur le treillis abstrait $\mathbb{B}_\perp$: si $v = \{0, 1\}$, on devra alors avoir $f(x) = v$, et si $v = \top$, alors la valeur de $f(x)$ est inconnue. Nous définissons donc formellement cette notion de représentation d'une fonction par un octree.

Définition 7.6 (Application d'un octree) Soit T un octree de dimension n et de domaine $D \in I_{\mathbb{R}}^n$, et soit $x \in D$. On note alors $T(x)$ la valeur associée à x dans l'octree T, définie par :

$$T(x) = v \text{ tel que } \exists\ N(b') \in T\ :\ x \in b' \text{ et } N(b') \text{ a un seul fils } V(v)\ .$$

Exemple 7.4 Soit T l'octree de la figure 7.1. On a alors : $T\big((3, 1.5)\big) = 1$ et $T\big((6, 1.2)\big) = \top$.

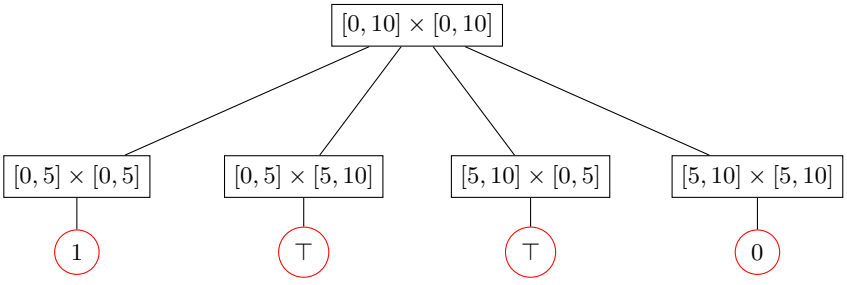
Remarque La complexité du calcul de l'application d'un octree à une valeur réelle est faible : le calcul peut clairement être effectué en un temps linéaire en la profondeur de l'octree, et donc logarithmique en la taille du domaine d'entrée.

Définition 7.7 (Représentation d'une fonction booléenne) Soit $D \in I_{\mathbb{R}}^n$ et $f : D \to \mathbb{B}$ une fonction booléenne. On dira qu'un octree T de dimension n est une représentation de f si et seulement si $\forall x \in D,\ f(x) \sqsubseteq T(x)$.

Clairement, il n'existe pas une unique représentation pour un fonction booléenne f. En particulier, si on note $dom(f) \subseteq \mathbb{R}^n$ le domaine de définition de f, alors l'octree $N\big(dom(f)\big) :: V(\top)$ est toujours une représentation de f. Pour distinguer deux représentations différentes d'une même fonction f, nous introduisons une relation de préordre sur l'ensemble des octrees qui est compatible avec la notion naturelle de précision de la représentation : on dira qu'une représentation est plus précise qu'une autre si elle renvoie moins souvent la valeur $\top$. Rappelons qu'un préordre est un ordre partiel ne disposant pas de la propriété d'antisymétrie.

Définition 7.8 (Précision de la représentation : préordre sur les graphes) Soit T et T' deux octrees avec $dom(T) = dom(T') = D$. On dira que T' est plus précis que T si et seulement si $\forall x \in D,\ T(x') \sqsubseteq T(x)$. On notera alors $T' \preceq T$. La relation $\preceq$ est un préordre sur l'ensemble des octrees de même domaine.

Exemple 7.5 L'octree de la figure 7.1 est plus précis que l'octree suivant :



Remarque Le préordre $\preceq$ n'est pas un ordre partiel car nous n'avons pas défini de forme normals sur les octrees. On peut en effet construire deux octree T et T' tels que pour tout x, $T(x) = T'(x)$ mais tels que T et T' ont des profondeurs différentes. On peut assez facilement définir une forme normale sur les octrees (entre deux octrees T et T' vérifiant $T \preceq T'$ et $T' \preceq T$, il suffit par exemple de prendre celui de profondeur minimale), et on peut alors utiliser toujours la forme normale ; le préordre $\preceq$ devient alors un ordre partiel. Nous ne détaillerons pas le mécanisme de normalisation des octree, mais dans notre implémentation, nous calculerons toujours les formes normales.

7.2.2 Algorithme d'abstraction d'une fonction booléenne

Nous montrons maintenant comment on peut construire un octree qui soit une représentation d'une fonction booléenne $\phi : D \to \mathbb{B}$, avec $D \in I_{\mathbb{R}}^n$. Pour cela, nous devons d'abord définir une fonction d'inclusion, ou fonction abstraite, pour ϕ, c'est-à-dire une fonction $\tilde{\phi} : I_{\mathbb{R}}^n \to \mathbb{B}_\perp$ qui soit une abstraction sûre de ϕ.

Définition 7.9 (Fonction d'inclusion booléenne) Soit $\phi : \mathbb{R}^n \to \mathbb{B}$ une fonction booléenne. On dira qu'une fonction $\tilde{\phi} : I_{\mathbb{R}}^n \to \mathbb{B}_\perp$ est une fonction d'inclusion pour ϕ si elle vérifie :

$$\forall b \in I_{\mathbb{R}}^n,\ \alpha\big(\{\phi(x)\ :\ x \in b\}\big) \sqsubseteq \tilde{\phi}(b) \tag{7.1}$$

où $\alpha : \mathcal{P}(\mathbb{B}) :\to \mathbb{B}_\perp$ est l'abstraction classique des ensembles de valeurs booléennes vers le treillis abstrait $\mathbb{B}_\perp$ et $\sqsubseteq$ est l'ordre partiel sur $\mathbb{B}_\perp$.

Exemple 7.6 Soit $\phi : \mathbb{R}^2 \to \mathbb{B}$ la fonction définie par $\forall x, y \in \mathbb{R}^2,\ \phi(x,y) = \begin{cases} 1 & \text{si } x = y \\ 0 & \text{sinon} \end{cases}$. La fonction $\tilde{\phi} : I_{\mathbb{R}}^2 \to \mathbb{B}_\perp$ définie par $\tilde{\phi}([x],[y]) = \begin{cases} 0 & \text{si } [x] \cap [y] = \emptyset \\ \top & \text{sinon} \end{cases}$ est une fonction d'inclusion pour ϕ.

Nous n'expliquons pas comment obtenir cette fonction d'inclusion pour toute fonction $\phi : \mathbb{R}^n \to \mathbb{B}$. Intuitivement, les calculs numériques effectués dans ϕ seront traduits en leurs équivalents sur le domaine des intervalles, et les tests seront surapproximés. Par exemple, la fonction $\phi(x) = x \leq 3$ sera traduite en : $\tilde{\phi}([x]) = \begin{cases} 1 & \text{si } \overline{x} \leq 3 \\ 0 & \text{si } \underline{x} > 3 \\ \top & \text{sinon} \end{cases}$. Nous expliquerons à la section 7.3 comment on peut construire cette fonction d'inclusion pour le cas qui nous intéresse, c'est-à-dire lorsque la fonction booléenne est une fonction de la forme : "étant donnée une entrée X, la ligne n du programme peut elle être exécutée ?". À l'aide d'une fonction d'inclusion $\tilde{\phi}$ pour une fonction $\phi : D \to \mathbb{B}$ avec $D \subseteq \mathbb{R}^n$, on peut construire un octree représentant ϕ.

Définition 7.10 (Calcul de la représentation d'une fonction booléenne) Soit $n \in \mathbb{N}$ et $D \in I_{\mathbb{R}}^n$ un domaine borné. Soit une fonction $\phi : D \to B$ et soit $\tilde{\phi} : I_{\mathbb{R}}^n \to \mathbb{B}_\perp$ une fonction d'inclusion pour ϕ. Soit $N_{max} \in \mathbb{N}$ une profondeur maximale. On définit alors T_ϕ l'octree de profondeur maximale N_{max} représentant ϕ par : $T_\phi =$ `ConstruireOctree`$(\tilde{\phi}, D, 0, N_{max})$, où `ConstruireOctree` est la fonction donnée par l'algorithme 3.

Remarque La fonction `ConstruireOctree` est décroissante en N_{max} : plus N_{max} est élevé, plus l'octree construit est précis.

Exemple 7.7 Soit $D = [0,10] \times [0,10] \subseteq \mathbb{R}^2$ et $\phi : D \to B$ la fonction définie par $\forall x, y \in D,\ \phi(x,y) = \big(d\big((x,y),(3.5,6)\big) \leq 1\big) \vee \big(d\big((x,y),(7,1.2)\big) \leq 1\big)$, où d est la fonction distance. Autrement dit, ϕ teste si un point (x,y) est à une distance inférieure à 1 du point $(3.5,6)$ ou du point $(7,1.2)$. L'algorithme de la définition 7.10 donne les octree de la figure 7.2 pour des précisions $N_{max} \in \{1,4,6,10\}$. Les parties vertes sont les zones où l'octree vaut 0, les zones jaunes celles où l'octree vaut 1, et les zones rouges signifient que l'octree renvoie $\top$.

Proposition 7.1 *Soit $D \subseteq \mathbb{R}^n$ et $\phi : D \to \mathbb{B}$ une fonction booléenne. Si $\tilde{\phi}$ est une fonction d'inclusion pour ϕ, alors pour tout $N_{max} \in \mathbb{N}$, $T =$ `ConstruuireOctree`$(\tilde{\phi}, D, 1, N_{max})$ est une représentation de ϕ, c'est-à-dire que pour tout $x \in D$, on a $\phi(x) \sqsubseteq T(x)$.*

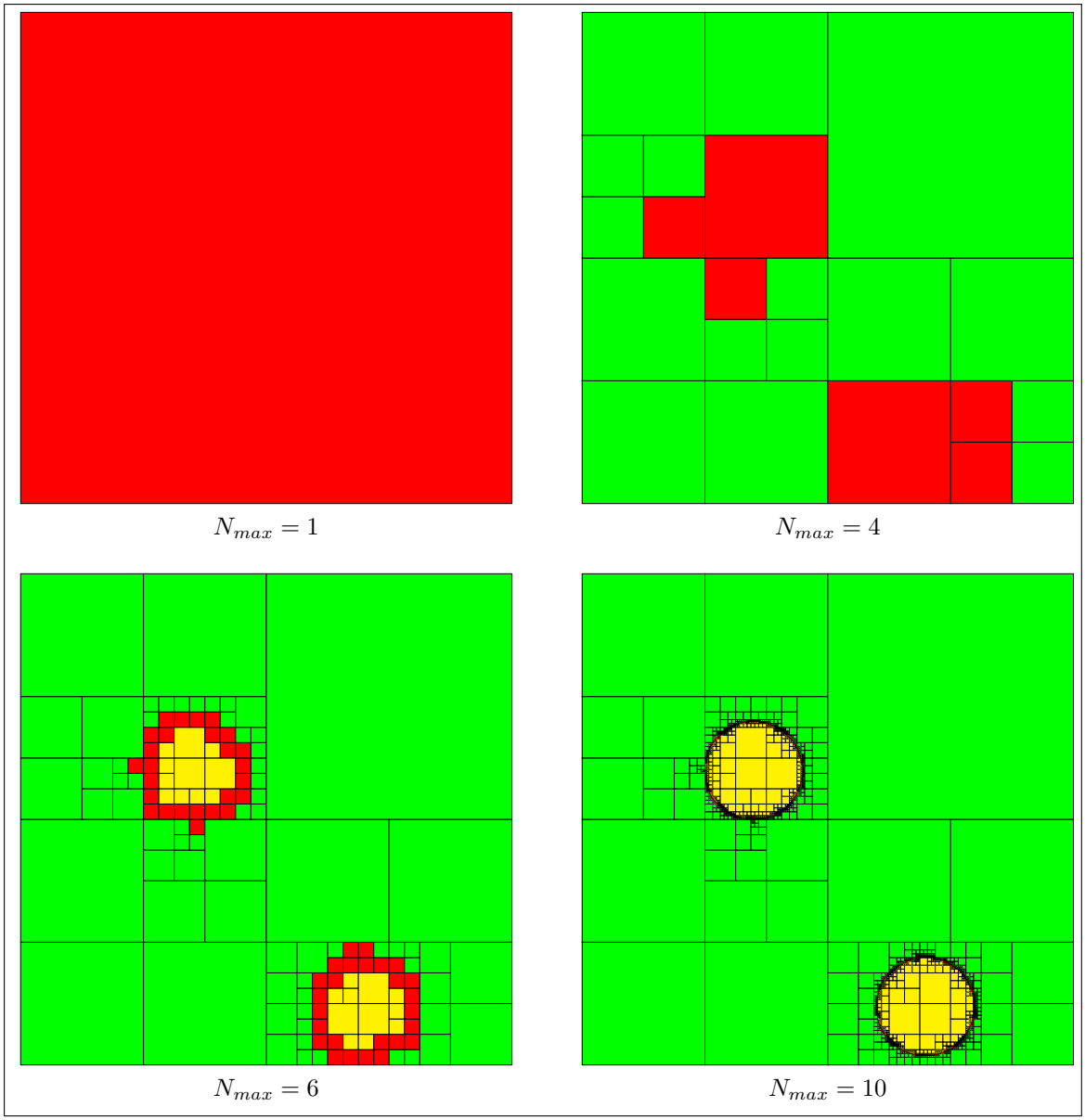


FIG. 7.2 – Résultats obtenus par l'algorithme `ConstruireOctree` pour la fonction de l'exemple 7.7.

Entrée : $\tilde{\phi} : I^n_{\mathbb{R}} \to \mathbb{B}_\perp$; `/* Fonction à abstraire */`
Entrée : $D \subseteq I^n_{\mathbb{R}}$; `/* Domaine de définition */`
Entrée : $N \in \mathbb{N}$; `/* Précision actuelle */`
Entrée : $N_{max} \in \mathbb{N}$; `/* Précision maximale */`
Résultat : T de domaine D et de hauteur maximale $N_{max} - N$ telle que
$\forall x \in D,\ \phi(x) \sqsubseteq T(x)$

début
  $x = \tilde{\phi}(D)$;
  si $x = \top \wedge N < N_{max}$ **alors**
    $list = Split(D)$;
    $res = N(D)$;
    pour *tout élément* $l \in list$ **faire**
      $aux =$ `ConstruireOctree`$(\tilde{\phi}, l, N+1, N_{max})$;
      $res = N(D) :: aux$;
    finpour
  sinon
    $res = N(D) :: V(x)$;
  finsi
  renvoyer res;
fin

Algorithme 3 : `ConstruireOctree`($\tilde{\phi}$,D,N,N_{max}) : fonction de construction d'un octree représentant une fonction ϕ.

7.3 Analyse d'accessibilité en utilisant des octree

Nous expliquons maintenant comment on peut utiliser l'algorithme de la section 7.2.2 pour une analyse de l'accessibilité de certaines instructions dans des programmes impératifs. Intuitivement, nous allons utiliser l'algorithme 3 avec pour fonction $\tilde{\phi}$ une abstraction de la fonction renvoyant 1 si l'instruction est exécutée et 0 sinon. La construction de cette fonction d'inclusion booléenne est donc le point le plus important de notre analyse d'accessibilité.

7.3.1 Analyse d'accessibilité

La fonction de test d'accessibilité est définie ainsi : étant donné un programme P et une instruction à atteindre, on construit dans un premier temps le graphe de flot de contrôle du programme puis on effectue une analyse statique par interprétation abstraite en avant des états accessibles. Autrement dit, on voit le programme comme un système de transitions discret et on calcule une surapproximation de l'ensemble des états accessibles.
Nous expliquons par un exemple comment cette analyse fonctionne. Soit le programme donné à la figure 7.3, avec son graphe de flot de contrôle (à droite sur la figure 7.3). Ce programme a donc deux entrées x et y que nous ne connaissons pas, avec $x \in [0, 10]$ et $y \in [0, 10]$. La figure 7.4 montre les états accessibles que l'on peut calculer lorsque $(x, y) = \big([0, 10], [0, 10]\big)$, $(x, y) = \big([0, 1], [0, 1]\big)$ et $(x, y) = \big([3, 4], [4, 5]\big)$. Nous avons à chaque fois décoré le graphe de flot de contrôle : les couleurs des états indiquent s'ils sont accessibles ou non, et les annotations associées aux états sont les valeurs des variables en ce point de contrôle. Nous donnons également les résultats des tests abstraits : par exemple, pour le test $d \leq 3$, $[\![d \leq 3]\!]$ représente le résultat de ce test lorsque d est un intervalle, et vaut donc 0 si le test est faux quelques soient les valeurs concrètes de d, 1 si il est vrai et $\top$ sinon.
Nous ne détaillons pas d'avantage le principe de l'analyse d'accessibilité, pour plus de détails sur ce sujet très classique, on pourra lire [CC77, CC94].

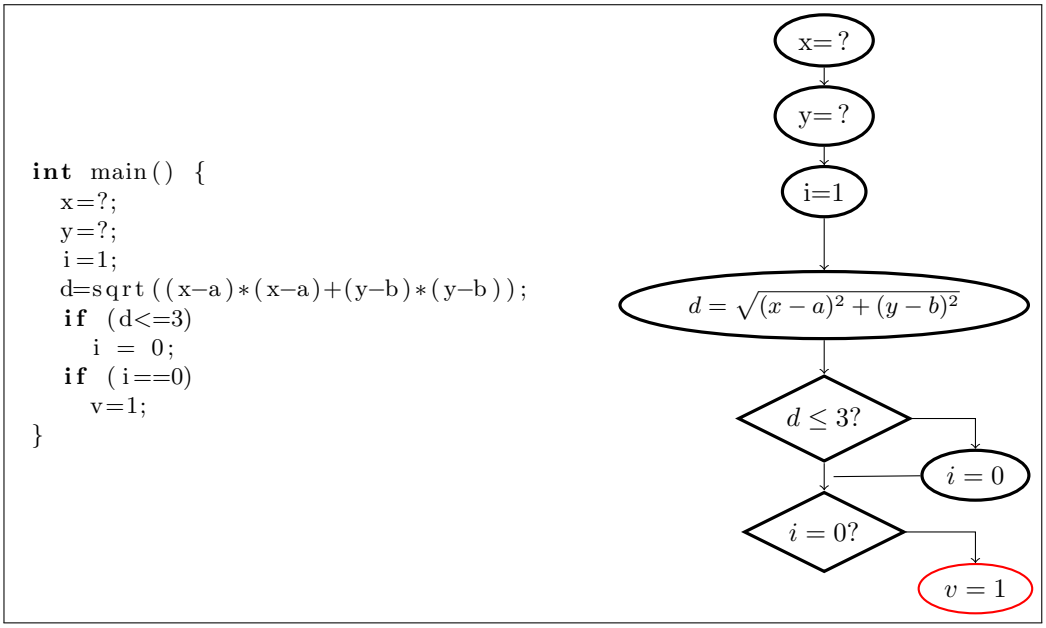


FIG. 7.3 – Un programme et son graphe de flot de contrôle. L'état rouge correspond au point de contrôle que l'on souhaite atteindre.

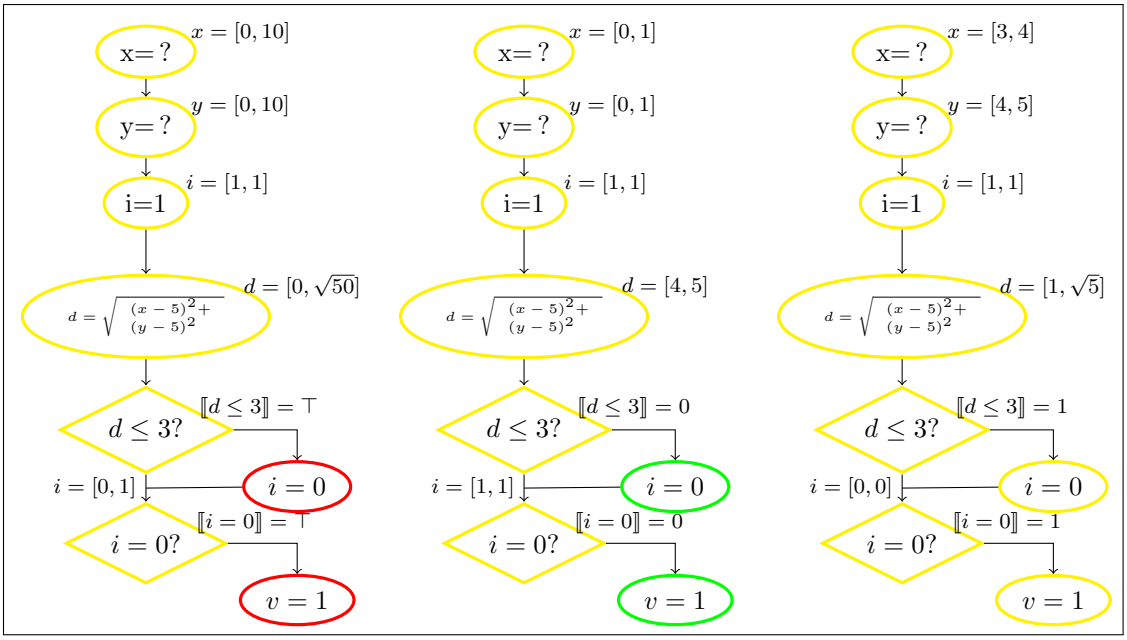


FIG. 7.4 – Trois analyses d'accessibilité pour des valeurs d'entrée différentes. Les états rouges sont les états potentiellement accessibles, les états verts les états que l'on ne peut pas atteindre et les états jaunes sont les états qui seront atteints quelques soient les valeurs d'entrées dans l'intervalle considéré.

Remarque Les travaux portant sur la terminaison de programmes et les preuves de propriétés de sûreté [CPR06a, CGP+07, Cou05] peuvent également servir à l'analyse d'accessibilité. De même, l'analyse de code mort [CGK97], souvent très utilisé en compilation, permet de prouver que certaines instructions du programme ne sont pas exécutées. Cependant, toutes ces analyses fournissent une surapproximation de l'ensemble des instructions exécutées. Pour que l'abstraction de l'évolution continue soit précise, il faut que nous obtenions également une sous-approximation des instructions accessibles, c'est-à-dire l'ensemble des instructions qui seront exécutées pour toutes les entrées du domaine. Dans l'idéal, la différence entre la surapproximation et la sous-approximation doit être la plus faible possible. Nous avons donc préféré développer un nouvel outil permettant de calculer à la fois la surapproximation et la sous-approximation de telle sorte que l'écart entre les deux soit un des paramètres de l'analyse.

7.3.2 Résultats expérimentaux

Nous avons développé un prototype d'analyseur qui calcule automatiquement un octree représentant la fonction d'accessibilité d'un programme écrit dans un sous langage de C contenant les opérations arithmétiques, les appels de fonctions, les différents types de boucles et de branchement conditionnels. Les tableaux et les pointeurs ne sont pour l'instant pas admis. Notre analyseur repose sur le logiciel Newspeak [HL08][1] pour l'analyse syntaxique des fichiers sources. Les entrées continues sont représentées par les variables globales et une syntaxe particulière de commentaires permet de définir le domaine initial. Par exemple, le programme de la figure 7.5 représente un programme avec deux variables continues (x et y), et les commentaires `/* !npk x=[0,10]*/` et `/* !npk y=[0,10]*/` indiquent que x et y prennent leurs valeurs dans $[0, 10]$. Notre analyseur est écrit OCAML et par soucis de performance nous avons utilisé un binding avec la libraire C++ Profil/BIAS pour le calcul par intervalle. L'analyseur peut produire un fichier XML représentant l'octree, l'afficher à l'écran ou encore produire un fichier PGF permettant son intégration dans un document LaTeX(les diagrammes de la figure 7.2 ont été obtenus ainsi).

Les figures 7.5, 7.6 et 7.7 montrent trois utilisations de cet analyseur. À chaque fois on donne le code source, l'octree calculé ainsi que les options utilisées (précision, ligne à atteindre) et le temps de calcul. Les tests ont été effectués sur un Intel Core 2 Duo à 1.66GHz avec 1Go de mémoire RAM. Sur les figures nous ne montrons que les zones où l'instruction demandée est atteinte (en jaune) et celles où l'octree vaut $\top$ (en rouge).

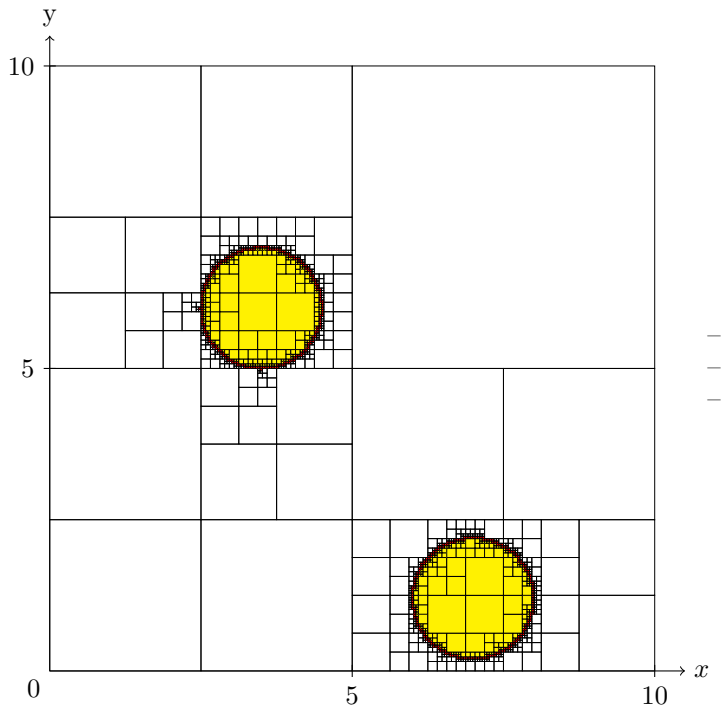
L'exemple de la figure 7.5 est celui dont on s'est servi pour obtenir les schémas de la figure 7.2. Il montre qu'il est possible d'obtenir facilement un recouvrement non convexe de l'espace. Les calculs effectués dans cet exemple sont non linéaires mais stables dans l'arithmétique des intervalles, de sorte que l'on peut obtenir rapidement une bonne précision : en utilisant une précision de 11 au lieu de 10, le temps de calcul passe à 0.20 secondes et la largeur maximale des rectangles rouges est de 10^{-2}.

L'exemple de la figure 7.6 est une adaptation du problème des deux réservoirs. On suppose que le mécanisme d'anticipation est donné par une extrapolation d'Euler à 0.5 secondes. On dispose donc de deux fonctions `F1` et `F2` qui donneent respectivement la valeur de la dérivée de x et de y. Ces fonctions représentent le cas où les deux vannes sont ouvertes, et un vrai contrôleur devra donc changer également ces fonctions en fonction des actionneurs. L'octree représente les zones de l'espace où la vanne 1 sera fermée (la ligne 40 représente l'effet du premier actionneur). On voit donc que l'on peut obtenir des ensembles de points ayant des formes non convexes et assez complexes, les branchements conditionnels dans les fonctions `F1` et `F2` causant des comportements très différents en fonction du lieu de x et y.

Enfin, l'exemple de la figure 7.7 montre que l'on peut également obtenir des résultats précis en présence de boucle dans le corps de la fonction `upd`. La fonction utilisée ici est une approximation de la fonction décidant si le point (x, y) appartient à l'ensemble de Mandelbrot. On voit donc apparaître une structure de fractale. Encore une fois, on peut obtenir une précision très grande mais le coût est très élevé à cause de l'instabilité numérique de la fonction `upd`.

[1] http ://www.penjili.org/newspeak.html

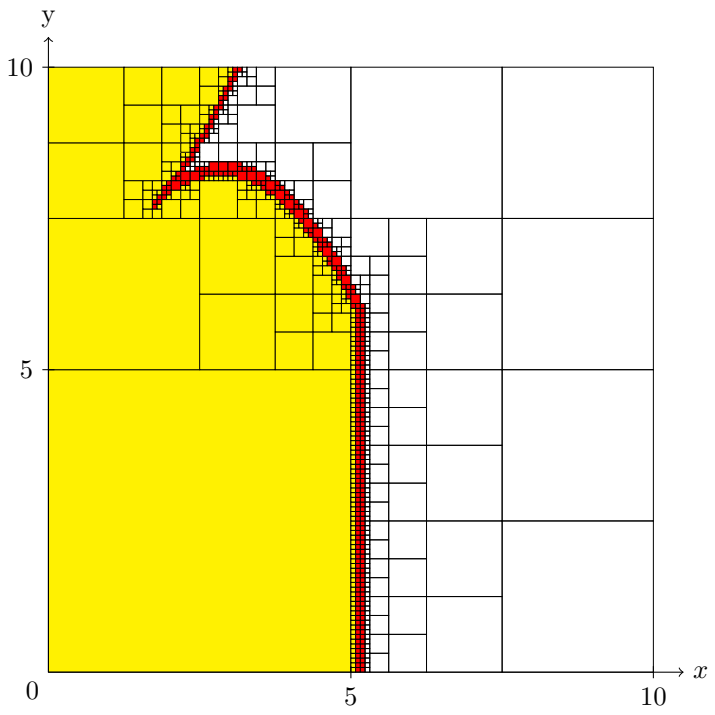
```
1  double x,y;
2  /*!npk x = [0,10] */
3  /*!npk y = [0,10] */
4
5
6  double dist (double x,double y,
7               double a,double b)
8  {
9    double res;
10   res = (x-a)*(x-a)
11     +(y-b)*(y-b);
12   return res;
13 }
14
15
16 void main() {
17   double res,res2;
18   int v;
19
20   res = dist (x,y,3.5,6);
21   res2 = dist (x,y,7,1.2);
22   if ((res<=1)||(res2<=1)) {
23     v=0;
24   }
25
26 }
```



- Ligne : 23
- Precision : 10
- Temps de calcul : 0.14s

FIG. 7.5 – Exemple des deux cercles.

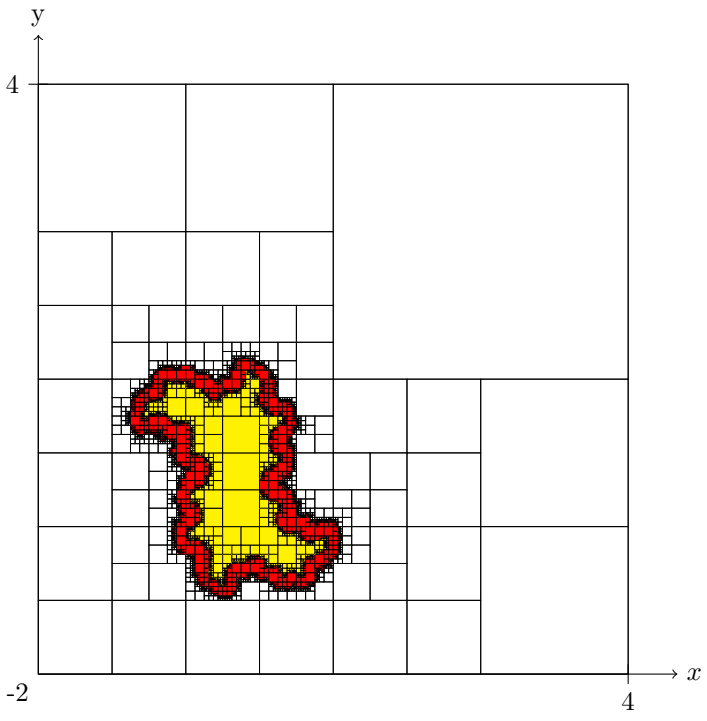
```
1  double x,y;
2  /*!npk x = [0,10] */
3  /*!npk y = [0,10] */
4
5  double H=6;
6  double k1=3,k2=6, i=1.5;
7
8  double F1(double x,double y) {
9    double sq1;
10   if (y>H)
11     sq1 = sqrt(x-y+H);
12   else
13     sq1 = sqrt(x);
14   return i-k1*sq1;
15 }
16
17 double F2(double x,double y) {
18   double sq1,sq2;
19   if (y>H) {
20     sq1 = sqrt(x-y+H);
21     sq2 = sqrt(y);
22   }
23   else {
24     sq1 = sqrt(x);
25     sq2 = sqrt(y);
26   }
27   return (k1*sq1 - k2*sq2);
28 }
29
30 void main() {
31   double h1_2,h2_2;
32   int v1, v2;
33
34   h1_2 = x+0.5*F1(x,y);
35   h2_2 = y+0.5*F2(x,y);
36
37   if (h1_2 > 8)
38     v1 = 1;
39   if (h1_2 < 2.5)
40     v1 = 0;
41   if (h2_2 > 9)
42     v2 = 1;
43   if (h2_2 < 2)
44     v2 = 0;
45 }
```



- Ligne : 40
- Precision : 10
- Temps de calcul : 0.24s

FIG. 7.6 – Exemple des deux réservoirs.

```
1  /*!npk x = [-2,4] */
2  /*!npk y = [-2,4] */
3  double x,y;
4
5
6  int upd (double x,double y)
7  {
8    double a,b,xt,yt,d;
9    int i = 0;
10   a = 0.32; b = 0.43; d = 0;
11   while (d<10) {
12     xt = x*x-y*y+a;
13     yt = 2*x*y + b;
14     x = xt;
15     y = yt;
16     d=x*x+y*y;
17     if (i>=10)
18       d = 30;
19     i++;
20   }
21   return i;
22 }
23
24 void main() {
25   int v = 0 , i = 0;
26   i = upd (x,y);
27   if (i>=11)
28     v = 1;
29
30 }
```



- Ligne : 28
- Precision : 9
- Temps de calcul : 7.60s

FIG. 7.7 – Exemple de fractales.

7.4 Application à l'analyse des systèmes hybrides

Nous donnons ici un exemple d'application de l'analyse d'accessibilité et de l'intégration garantie pour valider le comportement d'un système hybride. L'idée principale est la suivante : une fois que l'on a construit, pour chaque actionneur, l'octree correspondant aux zones dans lesquels il sera activé, on peut calculer une surapproximation du comportement continu du système. Pour cela, nous faisons une hypothèse simplificatrice qui est que l'on connaît le temps de discrétisation des capteurs, c'est-à-dire que les capteurs transmettent une valeur au programme régulièrement toutes les u secondes. Rappelons qu'étant donnée une fonction $F : \mathbb{R}^n \to \mathbb{R}^n$, une condition initiale $[y_0] \in I^n_{\mathbb{R}}$ et un temps final t, il existe une fonction $GRKLIB(F, [y_0], t)$ donnant un encadrement $[y] \in I^n_{\mathbb{R}}$ tel que toutes les solutions des problèmes de Cauchy de la forme $\dot{y} = F(y),\ y(0) = y_0$ avec $y_0 \in [y_0]$ vérifient $y(t) \in [y]$ (on trouvera la définition de cette fonction au chapitre 6). Soit donc κ un système hybride possédant m actionneurs. On calcule par une analyse d'accessibilité $2m$ octrees $T_1,\ T_2,\ \ldots, T_{2*m}$ représentant des zones $[b_1], \ldots, [b_{2*m}]$ de $\mathbb{R}^n$ telles que, si les capteurs donnent une valeur dans la zone b_i, un actionneur sera activé. Par convention, nous dirons que la zone $[b_i]$ avec $i \in [1, m]$ correspond à l'instruction $\boldsymbol{act}.i!1$ alors que la zone $[b_i]$ pour $i \in [m+1, 2m]$ correspond à l'instruction $\boldsymbol{act}.i!0$. Autrement dit, on construit des octrees $T_1, \ldots,\ T_{2m}$ tels que, pour tout $i \in [1, m]$, si les capteurs donnent une valeur y telle que $T_i(y) = 1$, alors l'instruction $\boldsymbol{act}.i!1$ du programme sera exécutée, alors que si $T_{m+i}(y) = 1$, c'est l'instruction $\boldsymbol{act}.i!0$ qui sera exécutée. On calcule alors une surapproximation de l'évolution continue grâce à l'algorithme 4. Nous expliquons le fonctionnement de cet algorithme avant de donner un exemple d'utilisation. Les entrées de l'algorithme sont d'une part une description du milieu continu (c'est-à-dire l'ensemble des modes continus disponibles représentés par un ensemble de fonctions F_k pour tout $k \in \mathbb{B}^m$), un état initial qui est un couple $[y_0] \in I^n_{\mathbb{R}}$, l'état initial continu, et $b \in \mathbb{B}^m$, le mode continu initial. La dernière entrée de l'algorithme est l'ensemble des octrees $T_1, \ldots,\ T_{2m}$ surapproximant l'effet du programme sur son environnement. L'algorithme construit alors un ensemble de couples $([y_n], b_n)$ pour tout $n \in \mathbb{N}$ tels que, à l'instant $n.u$, les capteurs fournissent une valeur $y \in [y_n]$ et l'évolution continue est régie par F_{b_n}. Pour cela, l'algorithme fonctionne ainsi : si on connaît $[y_n]$ et b_n pour un entier $n \in \mathbb{N}$ donné, on calcule $[y_{n+1}]$ grâce à la fonction $GRKLIB$, puis on teste la valeur de $T_i([y_{n+1}])$ pour chaque octree T_i ce qui permet de calculer b_{n+1}. Rappelons que, si on dispose d'un vecteur de booléen $b \in \mathbb{B}^m$, alors la notation $b[k \mapsto x]$ avec $k \in [1, m]$ et $x \in \{0, 1\}$ représente le vecteur $b' \in \mathbb{B}^m$ tel que toutes les coordonnées de b' sont égales à celles de b, sauf la k-ième qui vaut x.

Remarque (1) Nous ne considérons pas pour l'instant les cas où, que ce soit à cause d'une trop grande surapproximation ou par la faute du système lui même, on obtient $T_i([y_n]) = \top$ pour un encadrement $[y_n]$ et un octree T_i. Cela revient à dire qu'on ne peut pas décider quelle est la dynamique à utiliser pour calculer y_{n+1}, et cela introduit donc une divergence dans la suite des valeurs $[y_n]$. Dans une première approximation nous considérons que ces états sont des états potentiellement dangereux car ils peuvent causer une instabilité du programme embarqué en cas de capteur peu fiable, et nous lèverons donc un alarme dans ces circonstances. Une étude plus poussée de ces cas pathologiques pourrait apporter plus d'informations.

Remarque (2) L'algorithme fait la supposition que le programme contrôle efficacement l'environnement continu, c'est-à-dire que les valeurs fournies par les capteurs seront cycliques. Cependant, la terminaison théorique de cet algorithme n'est pas prouvée théoriquement et on peut juste supposer que ce sera le cas si le système hybride est stable. Pour obtenir la terminaison de l'algorithme, il faut effectuer des surapproximations supplémentaires en effectuant l'union de certains encadrement $[y_n]$ selon des heuristiques à définir, et éventuellement définir des opérateurs d'extrapolation pour accélérer la convergence.

7.4.1 Exemple d'applications : le système des deux réservoirs

Nous donnons maintenant un exemple de calcul de l'évolution continue par l'algorithme 4 pour le système des deux réservoirs (voir chapitre 1). Comme pour l'exemple de la figure 7.6, nous

Entrée : $\kappa = \big(Var, m, \{F_k\}_{k\in\mathbb{B}^m}\big)$; /* un modèle du système continu */
Entrée : $[y_0]$ et $b \in \mathbb{B}^m$; /* état initial du système */
Entrée : $\{T_n\}_{n\in[1,2m]}$; /* l'ensemble des octrees représentant le programme */
Résultat : $\big([y_n], b_n\big)_{n\in\mathbb{N}}$ avec $[y_n] \in I^n_{\mathbb{R}}$ et $b \in \mathbb{B}^m$

début
  $[y] = GRKLIB([y_0], F_b, u)$;
  $res = \{([y_0], b)\}$;
  tant que $([y], b) \notin res$ **faire**
    $res = res \cup ([y], b)$;
    pour $0 \leq i \leq 2m$ **faire**
      si $T_i([y]) = 1$ **alors**
        $b[m \mapsto 1]$;
      finsi
      si $T_{m+i}([y]) = 1$ **alors**
        $b[m \mapsto 0]$;
      finsi
    finpour
    $[y] = GRKLIB([y], F_b, u)$;
  fintantque
  renvoyer *res*;
fin

Algorithme 4 : Calcul des valeurs fournies par les capteurs.

supposerons que le contrôleur repose sur un mécanisme d'anticipation de l'état du système 0.5 secondes après l'instant présent, et que cette anticipation est effectuée par une méthode d'Euler simple. Dans le programme de la figure 7.6, l'anticipation n'est valable que pour le cas où les deux vannes sont ouvertes. Lorsque le programme agit sur un actionneur (représenté par l'instruction $v1 = 0$ par exemple), il faudra donc changer à la fois la dynamique continue et les fonction `F1` et `F2` permettant d'anticiper sur les hauteurs d'eau. Nous disposons donc de quatre programmes correspondant aux quatre valeurs possibles pour le couple $(v1, v2) \in \mathbb{B}^2$ dans le programme suivant de la figure 7.8.

Pour chacun des modes disponibles, c'est-à-dire pour chaque valeur possible du couple $(v1, v2)$, nous avons calculé les octrees correspondant aux différents actionneurs possibles. Cela donne les résultats de la figure 7.9. Ensuite, nous avons appliqué l'algorithme 4 pour calculer l'ensemble des valeurs prises par le milieu continu. Nous sommes partis de l'état initial suivant : les deux vannes sont ouvertes et $[x_0] = 7.38 + [-0.03, 0.03]$, $[y_0] = 6.67 + [-0.02, 0.02]$. Après 60 itérations, nous avons obtenu un point fixe, c'est-à-dire que l'évolution continue est revenue dans un état contenu dans l'état initial. La figure 7.10 montre la trajectoire des hauteurs d'eau au cours du temps. Le code des couleurs est le suivant : le rouge représente l'état "les deux vannes sont ouvertes", le bleu représente l'état "les deux vannes sont fermées", le vert représente l'état "la vanne 2 est fermée" et le noir représente l'état "la vanne 1 est fermée".

Remarque (1) Les résultat ont été obtenus de façon semi-automatique : le calcul du point d'approximation $[y_{n+1}]$ suivant et le calcul des valeurs $T_i([y_{n+1}])$ sont automatisés, mais le choix de la dynamique pour le pas suivant reste manuel. Il s'agit uniquement ici d'un problème d'implémentation et non d'un problème théorique.

Remarque (2) Le choix de la condition initiale a été effectué ainsi : nous avions initialement choisi une condition initiale exacte $(7, 7.5)$. Après une phase d'initialisation, le cycle de la figure 7.10 est apparu, mais sans que l'on obtienne de point fixe car les encadrements $[y_n]$ étaient de largeur très faible (environ 10^{-10}). Nous avons donc ajouté du bruit à un des points de ce cycle et recommencé l'analyse à partir de ce point, ce qui nous a permis d'obtenir un point fixe. D'autres

```
1  double x,y;
2  /*!npk x = [0,10] */
3  /*!npk y = [0,10] */
4
5  double H=6;
6  double k1=3,k2=6, i=1.5;
7
8  double F1(double x,double y) {
9    double sq1;
10   if (y>H)
11     sq1 = sqrt(x-y+H);
12   else
13     sq1 = sqrt(x);
14   return i-v1*k1*sq1;
15 }
16
17 double F2(double x,double y) {
18   double sq1,sq2;
19   if (y>H) {
20     sq1 = sqrt(x-y+H);
21     sq2 = sqrt(y);
22   }
23   else {
24     sq1 = sqrt(x);
25     sq2 = sqrt(y);
26   }
27   return (v1*k1*sq1 - v2*k2*sq2);
28 }
29
30 void main() {
31   double h1_2,h2_2;
32   int v1, v2;
33
34   h1_2 = x+0.5*F1(x,y);
35   h2_2 = y+0.5*F2(x,y);
36
37   if (h1_2 > 8)
38     v1 = 1;
39   if (h1_2 < 2.5)
40     v1 = 0;
41   if (h2_2 > 9)
42     v2 = 1;
43   if (h2_2 < 2)
44     v2 = 0;
45 }
```

FIG. 7.8 – Programme de contrôle du systèmes des deux réservoirs.

études avec d'autres conditions initiales sont en cours. Il est cependant impossible de prendre des conditions initiales très larges car la stabilité de la méthode d'intégration garantie dépend très fortement de la taille de l'encadrement initial.

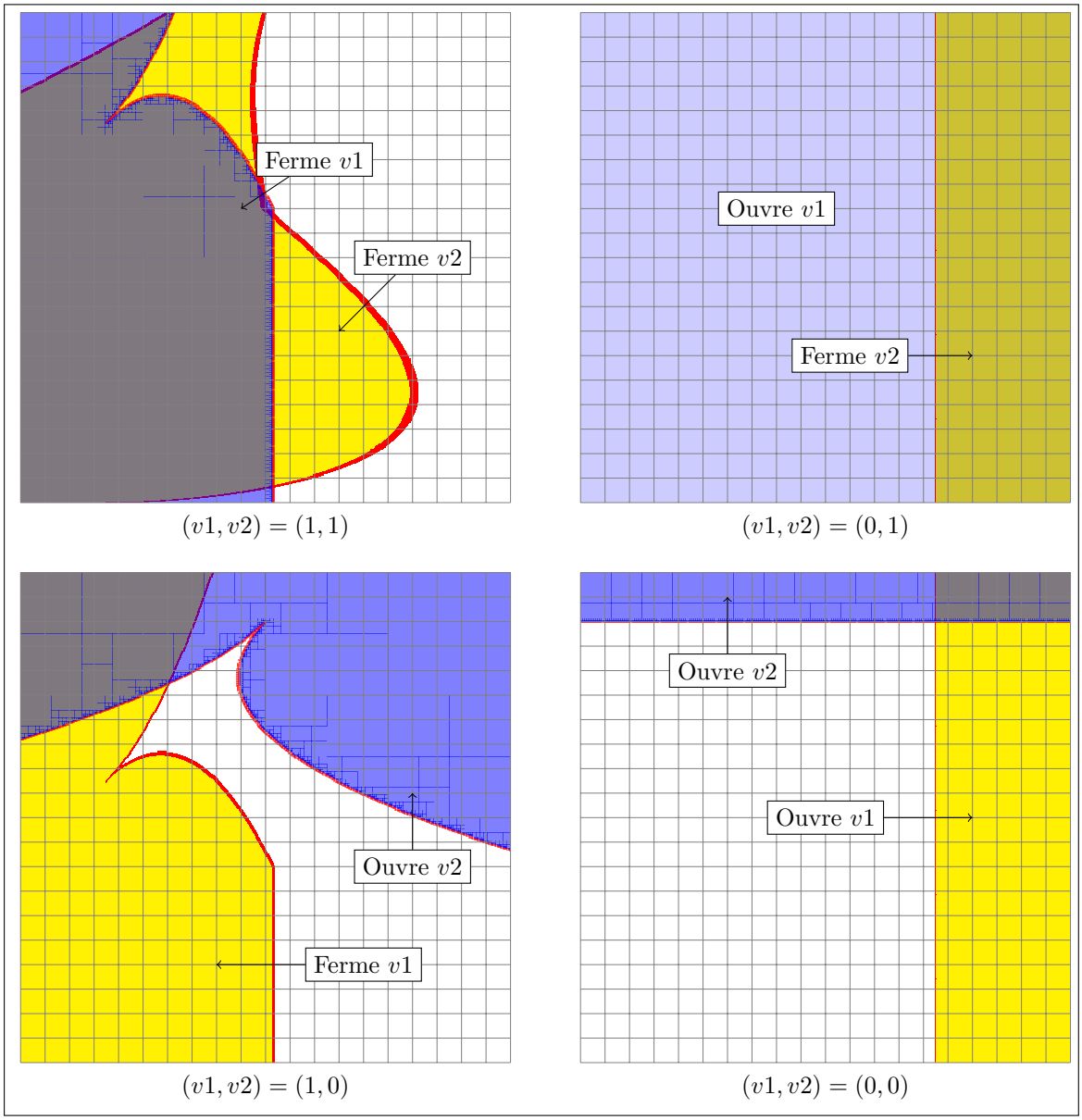


FIG. 7.9 – Octrees obtenus pour chaque mode du contrôleur des deux réservoirs.

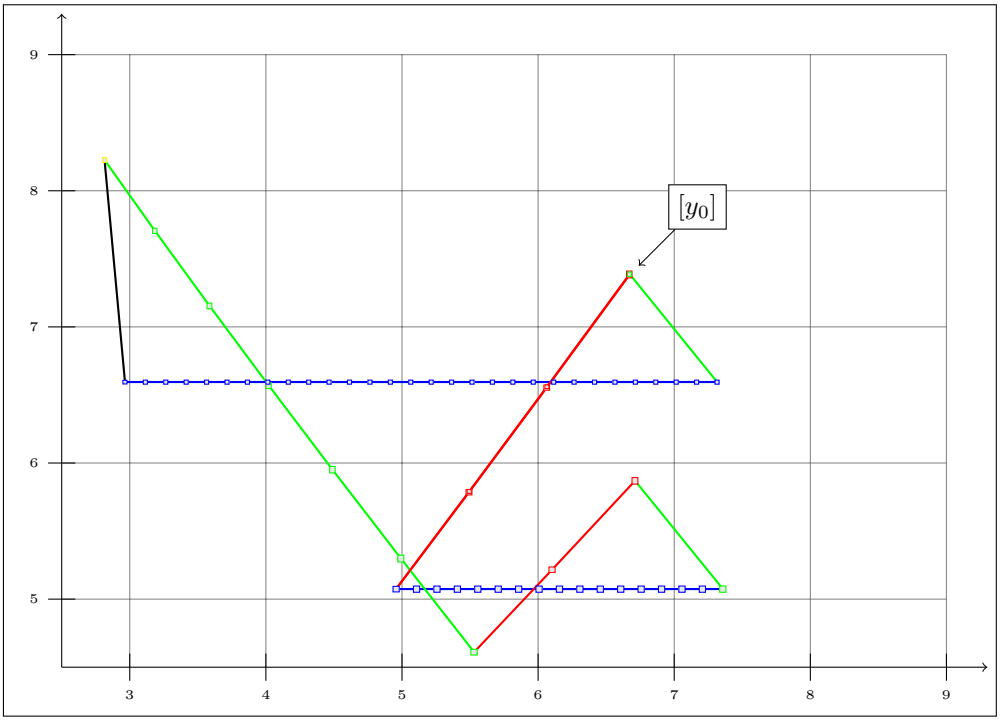


FIG. 7.10 – Évolution de x et y au cours du temps.

CHAPITRE 8

Conclusions et perspectives

8.1 Résumé du travail de thèse

Une extension des langages de programmation impératifs pour les systèmes hybrides Nous avons défini un nouveau modèle de systèmes hybrides qui se présente comme une extension des langages de programmation impératifs classiques. Ce modèle a été conçu dans le but de représenter et analyser les programmes critiques embarqués, et nous avons voulu qu'il soit le moins intrusif possible. Nous voulions donc qu'à la fois sa syntaxe et sa sémantique restent le plus proche possible des modèles discrets classiques.

Le modèle que nous proposons permet une séparation très forte des sous-systèmes discrets et continus : chaque partie est modélisée dans un formalisme différent (équations différentielles pour les modes continus, programme impératif pour le discret), et la seule contrainte que nous imposons est que les deux parties partagent la même interface, c'est-à-dire qu'ils s'accordent sur le nom des variables continues et le nombre d'actionneurs disponibles. Nous avons ensuite donné une sémantique dénotationnelle pour ce modèle. La sémantique s'appuie fortement sur la sémantique dénotationnelle classique des langages de programmation qu'elle étend. Pour cela, nous donnons une description de l'évolution continue (c'est-à-dire de la solution d'une équation différentielle) comme le point fixe d'un opérateur de Picard modifié défini sur le treillis complet des fonctions intervalles continues. Cet opérateur étant monotone, le point fixe existe et est calculable comme la limite des itérées de Kleene. On prouve alors que cette limite est bien la solution de l'équation différentielle. Nous intégrons ensuite ce calcul de l'évolution continue dans le calcul de la sémantique des programmes embarqués. La sémantique que nous définissons permet de prendre en compte la discrétisation temporelle et la discrétisation des valeurs continues en des nombres à virgule flottante qui intervient lorsque les capteurs transmettent une valeur au programme. Les actionneurs sont également pris en compte de telle sorte que la sémantique concrète de notre modèle décrit précisément l'évolution réelle des programmes embarqués. Ce modèle et cette sémantique nous sert ensuite de base pour une analyse plus complète et plus précise des programmes embarqués. Il n'existe à notre connaissance pas d'autres travaux ayant introduit dans la sémantique des programmes embarqués l'évolution de leur environnement continu.

Une nouvelle méthode d'intégration garantie Afin de calculer une abstraction de la sémantique de la partie continue nous avons défini une nouvelle méthode d'intégration garantie qui permet de calculer des bornes sûres sur la solution d'une équation différentielle. Contrairement à la majorité des travaux dans ce domaine, notre méthode repose sur un algorithme d'intégration numérique non validé (en l'occurrence l'algorithme de Runge-Kutta RK4) et nous représentons les encadrements

de la solution comme la somme d'un point non garanti (le point donné par RK4) et d'un intervalle d'erreur.

L'originalité de cette méthode vient de cette séparation entre nombre flottant et terme d'erreurs, ce qui permet de limiter le wrapping effect inhérent aux méthodes usuelles à base d'intervalles. Pour calculer le terme d'erreur, nous décomposons l'erreur globale après $n+1$ pas en trois : un terme qui représente l'erreur due à la méthode d'intégration numérique elle même, un terme représentant la propagation de l'erreur après n pas par l'équation différentielle elle-même, et enfin l'erreur due à l'utilisation de nombres en précision finie. Chaque source d'erreur est calculée indépendemment et des stratégies particulières permettent de les encadrer au mieux : on utilise la décomposition QR de Löhner pour l'erreur propagée et l'arithmétique dite de l'erreur globale pour l'erreur d'implémentation. Nous avons implémenté ces techniques dans une librairie C++ nommée GRKLib. Un gros travail d'implémentation a été effectué pour rendre GRKLib la plus efficace possible. En particulier, nous avons choisi de ne pas utiliser la différentiation automatique pour calculer les termes d'erreur mais une différentiation symbolique a priori. En effet, l'intérêt de la méthode GRKLib par rapport aux techniques classiques à base de séries de Taylor est que GRKLib n'a besoin que de la cinquième dérivée de la solution de l'équation différentielle pour calculer les termes d'erreur, alors que les outils comme VNODE utilisent au moins les 5 premières dérivées. Il est donc nettement plus avantageux dans notre cas de pré-calculer, symboliquement, la dérivée cinquième pour ne pas avoir à régénérer inutilement les quatre premières à chaque étape. Une pré-analyse de l'équation différentielle permet de faire cela. Les résultats obtenus, tant d'un point de vue stabilité de la méthode que du point de vue de l'efficacité, tendent à prouver l'intérêt de cette approche. En effet, pour une précision équivalente, GRKLib est nettement plus efficace qu'une méthode par séries de Taylor à l'ordre 5. De plus, le coût de chaque pas de calcul est très nettement réduit grâce au pré-calcul des dérivées.

Les bases d'une analyse globale du système hybride Enfin, nous avons posés les bases d'une analyse complète des programmes embarqués en interaction avec leur environnement physique. La grande difficulté vient encore une fois de la présence des actionneurs qui peuvent modifier la dynamique continue. L'analyse que nous proposons vise à casser la dépendance entre le programme et l'environnement continu en faisant deux analyses séparées.

La première analyse ne s'occupe que du programme et construit un recouvrement régulier de l'ensemble des entrées du programme de manière à identifier les zones b de l'espace telles que, si les valeurs fournies par les capteurs sont dans b, alors un actionneur sera activé. Pour construire ce recouvrement, nous utilisons des arbres réguliers (les *octree*) et des techniques d'analyse par intervalle combinée à une analyse d'accessibilité par interprétation abstraite classique. L'intérêt de cette approche est de pouvoir facilement augmenter la précision de l'analyse pour limiter au maximum les zones d'indécision, c'est-à-dire les ensembles de valeurs d'entrées pour lesquelles on ne sait pas si un actionneur sera activé ou non. Une grande précision est très importante pour avoir une analyse précise de l'évolution continue. En effet, la seconde analyse consiste à utiliser l'intégration garantie pour calculer une surapproximation de la suite de valeurs fournies par les capteurs au cours du temps. En partant des conditions initiales et en utilisant le recouvrement précédemment calculé, on peut obtenir une surapproximation de l'évolution continue pour tout $t \in \mathbb{R}_+$: on utilise GRKLib pour calculer une surapproximation sur un pas de temps, puis on teste grâce au partitionnement si un actionneur va être activé. Si oui, on utilise une autre dynamique pour calculer la surapproximation au pas de temps suivant ; si non, on continue avec la même dynamique.

Le problème restant est le cas où le partitionnement n'est pas assez précis et que les valeurs calculées par GRKLib sont dans une zone "rouge", c'est-à-dire que l'octree renvoie $\top$. On peut alors imaginer plusieurs solutions pour continuer l'analyse continue : soit on effectue un pas avec la dynamique actuelle, un pas avec la nouvelle, et on fait ensuite l'union des encadrements obtenus. Ceci risque malheureusement de rendre l'abstraction continue très peu précise en créant assez vite des encadrement très larges. On peut aussi effectuer une analyse disjonctive et garder les deux chemins possibles ; cette solution risque malheureusement d'être vite confrontée au problème de

l'explosion combinatoire. Il n'y a pas de solutions existantes qui soient pleinement satisfaisante tant d'un point de vue de la précision des résultats que de l'efficacité des calculs. Nous pensons qu'une analyse disjonctive avec une réduction du nombre de chemins par une relation d'équivalence approchée (c'est-à-dire que deux chemins seront équivalents s'ils sont suffisamment proches) peut être une bonne solution : on limiterait ainsi l'explosion combinatoire tout en contrôlant la précision.

Enfin, il nous reste à faire le lien entre l'analyse statique définie au chapitre 7 et la sémantique concrète définie au chapitre 5. Nous devons notament montrer que la dynamique abstraite calculée est une abstraction sûre de la dynamique concrète. Pour cela, il reste plusieurs étapes à franchir. Tout d'abord, nous devons montrer que l'octree construit est une abstraction sûre du programme discret. Cette preuve, quoique assez technique, ne nous semble pas présenter de difficultés majeures mais nous n'avons pas pu, par manque de temps, l'intégrer à cette thèse. Ensuite, il faut montrer que la suite de valeurs booléennes $\left\{b_n\right\}_{n\in\mathbb{N}}$ fournie par l'algorithme 4 est équivalente à la suite des changements de mode calculée par la sémantique concrète via la fonction $\sigma_\kappa.F$. Cette preuve est plus délicate à obtenir car la sémantique concrète est définie via un théorème de point fixe, mais le résultat précédent (l'octree est une abstraction sûre du programme) devrait aider. Enfin, avec ce résultat et les preuves de sûreté de GRKLib du chapitre 6, nous pourrons montrer que la suite des valeurs calculées par l'algorithme 4 est une abstraction sûre de la sémantique concrète de la partie continue définie au chapitre 5.

8.2 Perspectives

Nous présentons maintenant quelques problèmes qui restent ouverts à la fin de cette thèse, et qui sont de quatre ordres. Le travail théorique sur les modèles de système hybride doit être approfondi pour représenter au mieux les programmes embarqués. Le travail d'implémentation sur GRKLib doit être poursuivi pour gagner en stabilité et rapidité. Concernant l'analyse des systèmes hybrides, il est important de définir un widening temporel permettant d'analyser les programmes quelle que soit leur durée d'exécution. Enfin, nous souhaitons définir une analyse de certaines propriétés de vivacité pour les programmes embarqués.

8.2.1 Modèle théorique des programmes embarqués

Dans cette thèse, nous avons proposé un nouveau modèle de systèmes hybrides spécifique pour la modélisation et l'analyse des programmes embarqués. Notre modèle est moins général que les modèles classiques de systèmes hybrides (automates et calculs de processus hybrides), mais reste très expressif car il contient un langage de programmation complet. Il nous semble peu judicieux d'essayer de comparer le pouvoir expressif des automates hybrides et de notre modèle car les deux modèles sont construits sur des conceptions très différentes des processus hybrides (approche événementielle contre approche périodique, voir au chapitre 5). Les modèles hybrides les plus proches du notre sont des extensions des langages synchrones comme Modelica [OEM99] ou synERJY [BPS06]. Ces derniers sont des langages de haut niveau permettant de décrire de manière uniforme l'environnement continu et le contrôleur discret et leur sémantique semble assez proche de celle que nous avons donnée ; nous aimerions étudier plus en avant ces langages et notamment le code C qu'ils génèrent pour y appliquer nos outils d'analyse.

Par ailleurs, nous souhaitons étendre les caractéristiques hybrides que notre modèle peut traiter. Pour l'instant, nous avons ajouté aux programmes embarqués une description de leur environnement physique en faisant l'hypothèse que les capteurs et actionneurs étaient parfait, c'est-à-dire à action immédiate et n'introduisant aucun bruit. Un autre facteur peut modifier le comportement des programmes embarqués : l'architecture matérielle sur laquelle ils sont exécutés. Des travaux ont déjà été menés pour définir une sémantique des langages de programmation dépendante de l'architecture matérielle [NGC08]. La sémantique que nous avons définie peut être vue comme une sémantique des langages de programmation dépendante de l'environnement physique. Nous souhaitons construire une extension de **H-SIMPLE** qui permette de considérer ces deux aspects. Nous obtiendrions ainsi un modèle théorique permettant de décrire très précisément l'exécution d'un

programme embarqué sur une architecture matérielle donnée et dans un environnement physique donné. La prise en compte de l'environnement matériel nous permettrait également de considérer des capteurs imparfaits introduisant un bruit sur leurs résultats

Dans [NGC08], la sémantique opérationnelle des langages de programmation est étendue pour prendre en compte l'architecture matérielle. Pour cela, les auteurs considèrent un ensemble Π de fonctions décrivant les caractéristiques matérielles pouvant influer sur le comportement du programme : Π contient donc des informations quant à la taille de la représentation des nombres flottants et décrit comment sont effectuées les opérations d'alignement de structures ou de décalage. Dans le cas des programmes embarqués hybrides, nous pourrions ajouter à Π des fonctions de conversion entre les valeurs continues réelles et les valeurs discrètes flottantes. Cette conversion représente l'effet d'un capteur et peut donc introduire un certain bruit : pour modéliser un capteur ayant une précision de 10%, on affectera à la variable discrète une valeur aléatoire autour de la valeur réelle fournie par l'environnement continu. On disposerait alors d'un modèle $\left(\Delta, \kappa, \Pi\right)$ décrivant comment le programme (Δ) évolue dans un certain environnement continu (κ) et matériel (Π). On pourra alors prendre en compte lors de l'analyse les imprécisions dues aux capteurs et éventuellement les défaillances des actionneurs.

8.2.2 Extension de la librairie GRKLib

Comme nous l'avons mentionné, l'algorithme d'intégration garantie codé dans la librairie GRKLib est très générique et peut être appliqué à n'importe quelle méthode d'intégration numérique. Pour l'heure, nous l'avons appliqué à la méthode RK4 mais il est clair que pour concurrencer les méthode les plus avancées à base de série de Taylor (comme VNODE), il est nécessaire d'introduire dans la bibliothèque GRKLib d'autres méthodes numériques d'ordre supérieur. En utilisant des méthodes de Runge-Kutta d'ordre élevé, on peut donc raisonnablement supposer que l'efficacité et la précision de GRKLib seront grandement améliorées. Cependant, il n'est pas trivial d'un point de vue implémentation d'offrir à l'utilisateur un grand choix de méthodes garanties dans GRKLib. En effet, nous avons décidé de calculer les dérivées nécessaires au calcul des erreurs grâce à une différentiation symbolique lors de la phase de compilation. Nous utilisons actuellement pour cela les templates C++ mais la prochaine version de GRKLib écrite en OCAML devrait bénéficier de CAMLP4[1] pour effectuer la même chose. Ce choix, si il améliore grandement l'efficacité de notre méthode, rend également l'ajout de plusieurs méthodes à GRKLib plus complexe, car les dérivées à calculer ne sont pas forcément les mêmes pour toutes les méthodes. Il nous faudra donc pouvoir exprimer, pour chaque méthode, comment on peut obtenir les fonctions de calcul de l'erreur globale. Autrement dit, nous devrons écrire un outil de différentiation symbolique en méta-programmation (via les templates C++ ou via CAMLP4) qui puisse générer automatiquement le code des fonctions de calcul d'erreur, et un langage de description pour décrire ces fonctions.

Par ailleurs, l'expérience montre que sur la majorité des problèmes, l'élargissement du terme d'erreur est principalement dû au wrapping effect qui intervient lors du calcul de la propagation de l'erreur d'un pas sur l'autre. Pour réduire ce wrapping effect, on peut utiliser la technique de décomposition QR de Löhner si les termes d'erreur sont donnés par des intervalles, mais d'autres travaux reposent sur une représentation de l'erreur par des formes géométriques moins sensibles au wrapping effect. Ainsi, les modèles de Taylor ne souffrent que très peu de ce phénomène, mais restent très coûteux pour une utilisation réelle. Nous souhaitons étudier l'emploi des domaines numériques ayant fait leurs preuves en analyse statique de programmes (comme les polyèdres, octogones, zones...) pour réduire ce wrapping effect. Pour cela, il est nécessaire d'utiliser des domaines qui soient le plus stable possible par rotation : le wrapping effect intervient essentiellement lors du calcul de la propagation de l'erreur si la matrice de propagation est une matrice de rotation (voir chapitre 3). Pour réduire cet élargissement, il faut donc disposer d'un domaine sur lequel l'opération de rotation est la plus précise possible. Les polyèdres semblent adapté à cela, mais la complexité algorithmique des calculs peut s'avérer rédhibitoire dans un domaine (l'intégration garantie) où la vitesse des calculs comptent au moins autant que la précision. Un bon compromis pourrait être d'utiliser le domaine des zonotopes [Cox73] : un zonotope Z est un ensemble de points

[1] CAMLP4 est un préprocesseur pour OCAML qui permet notamment d'étendre facilement la syntaxe du langage. Voir `http ://brion.inria.fr/gallium/index.php/Camlp4`

qui peut s'exprimer en fonction d'un centre c et d'un ensemble de vecteurs $\{v_1,\ v_2, \ldots,\ v_n\}$:

$$Z = \left\{ c + \sum_{i=0}^{n} v_i a_i \ :\ \forall i \in [1,n],\ a_i \in [-1,1] \right\} .$$

Les zonotopes (récemment utilisés pour l'analyse des automates hybrides, [GG08]) offrent un bon compromis entre efficacité des calculs (même dans le cas de calculs non linéaires) et précision.

8.2.3 Un widening sur le temps

Un point très important sur lequel nous voulons également travailler est la question d'un *widening temporel*, c'est-à-dire un opérateur permettant, en fonction des n premières étapes de calcul, de surapproximer l'évolution continue jusqu'à $t = +\infty$. Cela est nécessaire pour étudier des programmes dont nous ne connaissons pas a priori la durée d'exécution. Dans le cas des équations différentielles classiques, les travaux de Lyapunov et de LaSalle permettent de prouver une stabilité asymptotique, c'est-à-dire permettent de calculer une région de l'espace vers laquelle le système converge lorsque t tend vers $+\infty$. Des extensions de ces résultats existent lorsque l'on considère des équations différentielles à commutateur (*switched differential equations*, [Hes04]) : on impose que la fonction de Lyapunov décroît aux points de discontinuité. Cependant, ces résultats sont utilisables pour le cas linéaire uniquement, où l'on connaît la forme de la fonction de Lyapunov et on peut donc la calculer automatiquement ; dans le cas non linéaire, le calcul de la fonction de Lyapunov est très difficilement automatisable. De plus, ces résultat ne fournissent pas de garantie quant à la position de l'évolution continue au cours du temps : ils prouvent uniquement qu'au bout d'un temps infini, la solution des équations différentielles sera dans une certaine région. On peut donc penser à utiliser ces résultat pour obtenir une sousapproximation de l'extrapolation sur le temps, mais ils ne peuvent pas définir un widening temporel.

Remarquons que dans le cas d'équations différentielles linéaires, et si les zones de l'espace sont représentées par des polyèdres, on sait calculer une surapproximation de l'évolution continue (c'est ce que est utilisé pour la vérification des automates hybrides linéaires, voir chapitre 4). Dans le cas non linéaire, tout est évidemment plus compliqué. Une piste potentiellement intéressante est d'essayer de détecter un motif dans l'évolution continue. Remarquons en effet que les systèmes auxquels nous nous intéressons sont des systèmes *invariants dans le temps* : les équations différentielles sont autonomes et les programmes vérifient la propriété d'invariance dans le temps décrite au chapitre 7. Ainsi, si l'évolution continue est cyclique, alors on sait que les variables continues ne quitteront jamais ce cycle et on peut donc l'utiliser comme extrapolation à l'infini. Il suffit alors d'utiliser le domaine des congruences [Gra91] pour le temps pour avoir une surapproximation du comportement continu jusqu'à un temps infini. Si aucun cycle n'apparaît, on peut essayer de détecter dans l'évolution continue une progression selon une suite arithmético-géométrique. Nous nous inspirons pour cela des techniques développées par Jérôme Feret [Fer05] pour l'analyse des codes critiques embarqués : le domaine des progressions arithmético-géométrique relie les valeurs des variables à l'intérieur d'une boucle avec un compteur i qui décrit le nombre d'itérations de la boucle. La valeur d'une variable après i itérations est de la forme $f^i(M)$ où M est une valeur initiale et f est une fonction de la forme $f(X) = \alpha X + \beta$, avec $\alpha, \beta \in \mathbb{R}$. Dans notre cas, on essaiera de relier la valeur des variables avec le temps d'exécution du programme.

Le domaine des progressions arithmético-géométriques ne permet cependant que de détecter des progressions relativement simples pour les variables continues. Si les équations différentielles sont fortement non-linéaires, les surapproximations risquent d'être trop importantes et la précision trop faible. Une autre solution est alors d'utiliser les chaînes de récurrences [BWZ94, Pop06], qui sont une extension des progressions arithmético-géométriques. Une chaîne de récurrence est une représentation d'une fonction G comme une suite de termes φ et d'opérations $\odot$ que nous noterons $\Phi = \{\varphi_0, \odot_1, \varphi_1, \odot_1, \ldots, \odot_k, \varphi_k\}$. Cette représentation permet de calculer efficacement l'ensemble des termes de la forme $G(x_0 + i * h)$ pour $i \in \mathbb{N}$. Par exemple, la fonction $G(x) = e^{x^2}$ est représentée par la chaîne de récurrence $\Phi = \{e^{x_0^2}, *, e^{2x_0h+h^2}, *, e^{2h^2}\}$. Pour calculer $G(x_0 + i * h)$, il faut alors évaluer Φ en i ; nous ne détaillons pas comment l'évaluation de Φ est définie, mais elle est très

rapide si on a la valeur de $\Phi(i-1)$ (voir [BWZ94] pour plus de détails). Les chaînes de récurrences ont été utilisées en compilation pour déterminer à l'avance le nombre d'itérations dans une boucle [Pop06]. Les motifs de la chaîne de récurrence sont calculés dynamiquement en fonction des valeurs successives prises par la fonction G en x_0, x_0+h, $x_0+2h, \ldots$ Les évolutions que l'on peut obtenir sont plus variées qu'avec le domaine des progressions arithmetico-géométriques et on pourrait, en utilisant ces chaînes de récurrence, prédire la position des variables continues pour tout instant de la forme $t_0 + i * h$, où h est l'intervalle de temps entre deux mesures et i l'indice de la mesure.

En utilisant ces différentes représentations (suite cyclique, domaine des progressions arithmético-géométrique, chaîne de récurrence), on peut donc représenter de façon finie l'ensemble des valeurs de la forme $y(t_0 + i * h)$ où y est la fonction donnant la valeur des variables continues en fonction du temps. La preuve de la sûreté de cette représentation nous semble cependant loin d'être triviale car la fonction y n'est pas connue explicitement mais est définie comme la solution d'une équation différentielle.

8.2.4 Preuve de vivacité des systèmes hybrides

Les analyses que nous avons présentées dans cette thèse permettent avant tout d'obtenir des invariants plus précis pour les valeurs du programme en considérant l'environnement physique avec lequel il interagit. Ceci permettrait donc d'améliorer les preuves de propriétés de sûreté des programmes embarqués (comme l'absence de RTE par exemple). Cependant, pour les programmes embarqués, prouver ces propriétés ne nous semble pas être suffisant pour garantir le bon fonctionnement d'un système. Prenons en effet l'exemple des erreurs de calcul dues à l'utilisation de nombre à virgule flottante que calcule l'outil Fluctuat. Le lien entre l'erreur de calcul et le mauvais fonctionnement du programme est peu évident : on peut très bien imaginer des programmes dont l'erreur de calcul soit très élevée mais tel que cette erreur ne gêne pas le fonctionnement du programme. La notion de comportement que nous utilisons ici est une vue fonctionnelle du programme : on voit un programme comme une boîte noire dont le but est de contrôler les variables continues avec lesquelles il interagit. Dans cette optique, prouver des propriétés de sûreté sur le programme est certes nécessaire mais ne constitue pas une fin en soi : ce qui compte n'est pas tant que le programme ne fasse pas d'erreurs mais qu'il contrôle bien les variables qu'il est censé contrôler. Après une première analyse des propriétés de sûreté nous voulons donc mettre en place une analyse de propriétés de vivacité.

Les propriétés que nous souhaitons prouver sont des propriétés de la forme : si le programme reçoit une valeur X (via les capteurs), alors avant un délai t, l'actionneur m sera activé. Ce genre de propriétés se décrit généralement par une formule de logique linéaire temporelle (*linear temporal logic*, ou LTL) et c'est souvent dans cette logique que sont exprimées les propriétés à prouver pour la vérification des automates hybrides [HM00]. Pour la vérification des programmes, il faudra sûrement utiliser une variante du μ-calcul [Koz83] comme dans [CC00] et [Mas02]. Détaillons pour finir la propriété principale que nous aimerions pouvoir prouver. On voit un programme embarqué comme une boîte noire et on ne s'intéresse qu'aux transitions entre cette boîte noire et l'environnement extérieur. On peut donc voir l'exécution du programme dans son environnement comme une suite d'actions (capteurs et actionneurs) effectuées à des temps déterminés. Selon que le programme utilise des nombres réels ou des nombres flottants, cette suite d'actions peut être légèrement modifiée (voir figure 8.1). Nous voulons alors montrer que malgré les erreurs de calcul flottant, les deux suites sont suffisamment proches, c'est-à-dire qu'il existe un temps τ tel que, si le programme exécuté avec des nombres réels effectue une action à un instant t, alors le même programme exécuté avec des nombres à virgule flottante effectuera la même action à un instant $t' \in [t-\tau, t+\tau]$. On cherche donc une sorte de bisimulation faible approchée entre le programme utilisant des nombres réels et le programme utilisant des nombres à virgule flottante : peu importe si leur fonctionnement interne est très différent, on ne peut pas *observer* de différences notables entre les deux.

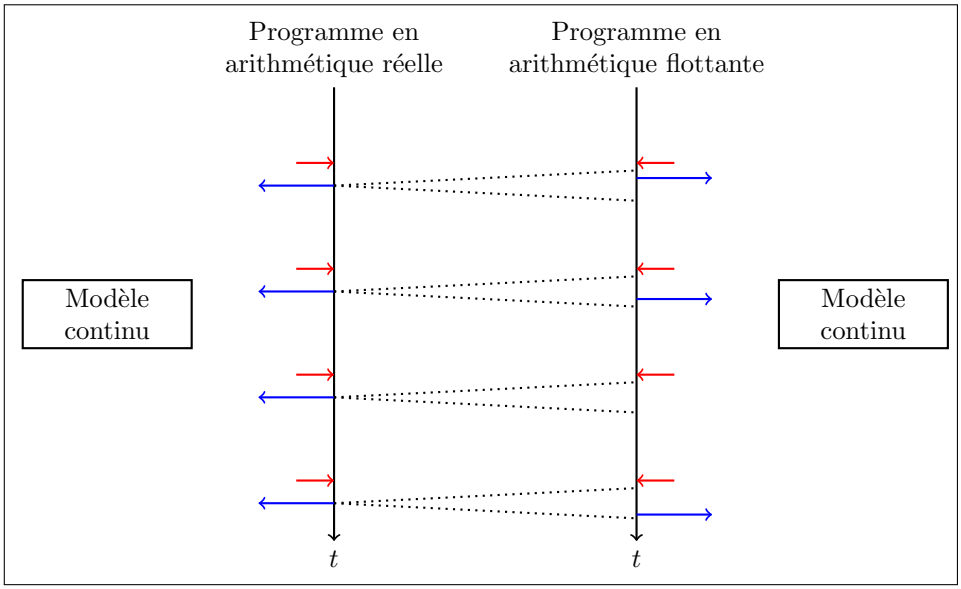


FIG. 8.1 – Critère de sûreté sur les programmes embarqués. À chaque action qu'effectuerait le programme en utilisant des nombres réels doit correspondre une action effectuée par le programme avec des nombres à virgule flottante à un temps proche. Les flèches rouges correspondent aux données transmises par les capteurs, les flèches bleues sont les commandes envoyées par les actionneurs. Dans cet exemple, le programme n'est pas sûr car la troisième action n'est pas exécutée en arithmétique flottante alors qu'elle aurait due l'être.

Bibliographie

[ACH+95] R. Alur, C. Courcoubetis, N. Halbwachs, T. A. Henzinger, P. H. Ho, X. Nicollin, A. Olivero, J. Sifakis, and S. Yovine. The algorithmic analysis of hybrid systems. *Theoretical Computer Science*, 138(1) :3–34, 1995.

[ACHH92] R. Alur, C. Courcoubetis, T. A. Henzinger, and P. H. Ho. Hybrid automata : An algorithmic approach to the specification and verification of hybrid systems. In *Hybrid Systems I*, volume 736 of *Lecture Notes in Computer Science*, pages 209–229. Springer, 1992.

[AD94] R. Alur and D.L. Dill. A theory of timed automata. *Theoretical Computer Science*, 126 :183–235, 1994.

[AGG08] X. Allamigeon, S. Gaubert, and E. Goubault. Inferring Min and Max Invariants Using Max-plus Polyhedra. In *Proceedings of the 15th International Static Analysis Symposium (SAS'08)*. Springer Verlag, 2008. To appear.

[Ant00] P.J. Antsaklis. A brief introduction to the theory and applications of hybrid systems. In *Special issue on hybrid systems : theory and applications*, volume 88 of *Proceedings of the IEEE*, pages 879–887. IEEE, 2000.

[BCC+03] B. Blanchet, P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, D. Monniaux, and X. Rival. A static analyzer for large safety-critical software. In *Proceedings of the ACM SIGPLAN 2003 conference on Programming language design and implementation (PLDI'03)*, pages 196–207. ACM, 2003.

[Ber05] J. Bertrane. Static analysis by abstract interpretation of the quasi-synchronous composition of synchronous programs. In *Proceedings of the 6th International Conference on Verification, Model Checking and Abstract Interpretation (VMCAI'05)*, volume 3385 of *Lecture Notes in Computer Science*, pages 97–112. Springer, 2005.

[BH98] M. Berz and G. Hoffstätter. Computation and application of taylor polynomials with interval remainder bounds. *Reliable Computing*, 4(1) :83–97, 1998.

[BHN02] C. Bischof, P. Hovland, and B. Norris. Implementation of automatic differentiation tools. *SIGPLAN Notices*, 37(3) :98–107, 2002.

[Bie51] L. Bieberbach. On the remainder of the Runge-Kutta formula. *Zeitschrift für angewandte Mathematik und Physik*, 2 :233–248, 1951.

[Bir67] G. Birkhoff. *Lattice Theory (3rd ed.)*. Amer. Math. Soc. Colloquium Publications, 1967.

[BLARS07] I. Bogudlov, T. Lev-Ami, T. Reps, and M. Sagiv. Revamping TVLA : Making parametric shape analysis competitive. In *19th International Conference on Computer Aided Verification (CAV'07)*, volume 4590 of *Lecture Notes in Computer Science*, pages 221–225, 2007.

[BM98] M. Berz and K. Makino. Verified integration of odes and flows using differential algebraic methods on high-order taylor models. *Reliable Computing*, 4(4) :361–369, 1998.

[BM06a] M. Berz and K. Makino. COSY INFINITY Version 9. *Nuclear Instruments and Methods*, A558 :346–350, 2006.

[BM06b] O. Bouissou and M. Martel. GRKLib : a guaranteed runge-kutta library. In *International Symposium on Scientific Computing, Computer Arithmetic and Validated Numerics*. IEEE, 2006.

[BM07a] O. Bouissou and M. Martel. GRKLib : a guaranteed runge-kutta library. In *Follow-up of International Symposium on Scientific Computing, Computer Arithmetic and Validated Numerics*. IEEE Press, 2007.

[BM07b] O. Bouissou and M. Martel. Static analysis of embedded programs with continuous i/o (poster). In *Hybrid Systems, Computation and Control (HSCC'07)*, 2007.

[BM08a] O. Bouissou and M. Martel. Abstract interpretation of the physical inputs of embedded programs. In *Proceedings of the 9th International Conference on Verification, Model Checking and Abstract Interpretation (VMCAI'08)*, volume 4905 of *Lecture Notes in Computer Science*, pages 37–51. Springer, 2008.

[BM08b] O. Bouissou and M. Martel. A hybrid denotational semantics of hybrid systems. In *Proceedings of the 17th European Symposium on Programming Languages and Systems (ESOP'08)*, volume 4960 of *Lecture Notes in Computer Science*, pages 63–77. Springer, 2008.

[Boa96] Ariane 5 Inquiry Board. Ariane 5 – flight 501 failure. Technical report, European Space Agency, 1996. http ://esamultimedia.esa.int/docs/esa-x-1819eng.pdf.

[BPS06] Reinhard Budde, Axel Poigné, and Karl-Heinz Sylla. synERJY an object-oriented synchronous language. *Electr. Notes Theor. Comput. Sci.*, 153(4) :99–115, 2006.

[BRZH02] R. Bagnara, E. Ricci, E. Zaffanella, and P. Hill. Possibly not closed convex polyhedra and the parma polyhedra library. In *Proceedings of the 9th International Symposium on Static Analysis (SAS'02)*, volume 2477 of *Lecture Notes in Computer Science*, pages 213–229. Springer-Verlag, 2002.

[BS96] C. Bendtsen and O. Stauning. FADBAD, a flexible c++ package for automatic differentiation using the forward and backward methods. Technical Report IMM–REP–1996–17, Department of Mathematical Modelling, Technical University of Denmark, 1996.

[BW90] J.C.M. Baeten and W.P. Weijland. *Process algebra*. Cambridge University Press, New York, NY, USA, 1990.

[BWZ94] O. Bachmann, P. Wang, and E. Zima. Chains of recurrences—a method to expedite the evaluation of closed-form functions. In *International symposium on Symbolic and algebraic computation (ISSAC'94)*, pages 242–249, New York, NY, USA, 1994. ACM.

[Car58] J. W. Carr. Error bounds for the Runge-Kutta single-step integration process. *Journal of the ACM*, 5(1) :39–44, 1958.

[CC77] 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 Fourth ACM Symposium on Principles of Programming Languages (POPL'77)*, pages 238–252. ACM Press, 1977.

[CC92a] P. Cousot and R. Cousot. Abstract interpretation frameworks. *Journal of Logic and Computation*, 2(4) :511–547, 1992.

[CC92b] P. Cousot and R. Cousot. Comparing the Galois connection and widening/narrowing approaches to abstract interpretation, invited paper. In *Proceedings of the*

4th International Workshop Programming Language Implementation and Logic Programming (PLILP'92), volume 631 of *Lecture Notes in Computer Science*, pages 269–295. Springer-Verlag, 1992.

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

[CC00] P. Cousot and R. Cousot. Temporal abstract interpretation. In *Conference Record of the Twentyseventh Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 12–25, Boston, Mass., January 2000. ACM Press, New York, NY.

[CCF+05] P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Minéand D. Monniaux, and X. Rival. The ASTREÉ analyzer. In *Proceedings of the 14th European Symposium on Programming (ESOP'05)*, volume 3444 of *Lecture Notes in Computer Science*, pages 21–30. Springer, 2005.

[CGK97] Y. Chen, E. Gansner, and E. Koutsofios. A C++ data model supporting reachability analysis and dead code detection. In *Proceedings of the 6th European Software Engineering Conference (ESEC/FSE 97)*, volume 1301 of *Lecture Notes in Computer Science*, pages 414–431. Springer, 1997.

[CGP+07] B. Cook, A. Gotsman, A. Podelski, A. Rybalchenko, and M. Vardi. Proving that programs eventually do something good. *SIGPLAN Notices*, 42(1) :265–276, 2007.

[CH78] P. Cousot and N. Halbwachs. Automatic discovery of linear restraints among variables of a program. In *Conference Record of the Fifth ACM Symposium on Principles of Programming Languages (POPL'78)*, pages 84–97. ACM Press, 1978.

[Cou05] P. Cousot. Proving program invariance and termination by parametric abstraction, lagrangian relaxation and semidefinite programming. In *Proceedings of the Sixth International Conference on Verification, Model Checking and Abstract Interpretation (VMCAI'05)*, volume 3385 of *Lecture Notes in Computer Science*, pages 1–24. Springer, 2005.

[Cox73] H.S.M. Coxeter. *Regular Polytopes*. Dover Publications, 1973.

[CPR06a] B. Cook, A. Podelski, and A. Rybalchenko. Termination proofs for systems code. In *Proceedings of the 2006 ACM SIGPLAN conference on Programming language design and implementation (PLDI'06)*, pages 415–426. ACM, 2006.

[CPR06b] B. Cook, A. Podelski, and A. Rybalchenko. Terminator : Beyond safety. In *18th International Conference on Computer Aided Verification (CAV'06)*, volume 4144 of *Lecture Notes in Computer Science*, pages 415–418. Springer, 2006.

[DHL+05] D. Daney, G. Hanrot, V. Lefèvre, P. Pélissier, F. Rouillier, and P. Zimmermann. The MPFR library, 2005.

[DS07] D. Delmas and J. Souyris. Astrée : From research to industry. In Hanne Riis Nielson and Gilberto Filé, editors, *Proceedings of the 14th International Symposium on Static Analysis (SAS'07)*, volume 4634 of *Lecture Notes in Computer Science*, pages 437–451. Springer, 2007.

[Eij81] P. Eijgenraam. *The Solution of Initial Value Problems Using Interval Arithmetic*. Mathematical Centre Tracts No. 144. Stichting Mathematisch Centrum, Amsterdam, 1981.

[EL02] A. Edalat and A. Lieutier. Domain theory and differential calculus. *Mathematical Structures in Computer Science*, 14(06) :771–802, 2002.

[EP04] A. Edalat and D. Pattinson. A domain theoretic account of Picard's theorem. In *31st International Colloquium on Automata, Languages and Programming (ICALP'04)*, volume 3142 of *Lecture Notes in Computer Science*, pages 494–505. Springer, 2004.

[Fer04] J. Feret. Static analysis of digital filters. In *Proceedings of the 13th European Symposium on Programming Languages and Systems (ESOP'04)*, volume 2986 of *Lecture Notes in Computer Science*, pages 33–48. Springer, 2004.

[Fer05] J. Feret. The arithmetic-geometric progression abstract domain. In *Proceedings of the 6th International Conference on Verification, Model Checking and Abstract Interpretation (VMCAI'05)*, volume 3385 of *Lecture Notes in Computer Science*, pages 42–58. Springer, 2005.

[FHL+01] C. Ferdinand, R. Heckmann, M. Langenbach, F. Martin, M. Schmidt, H. Theiling, S. Thesing, and R. Wilhelm. Reliable and precise WCET determination for a real-life processor. In *First International Workshop on Embedded Software (EMSOFT'01)*, volume 2211 of *Lecture Notes in Computer Science*, pages 469–485. Springer, 2001.

[FM92] K. Forsberg and H. Mooz. The relationship of systems engineering to the project cycle. *Engineering Management Journal*, 4(3) :36–38, 1992.

[Fre05] G. Frehse. Phaver : Algorithmic verification of hybrid systems past hytech. In Manfred Morari and Lothar Thiele, editors, *Proceedings of the , 8th International Workshop on Hybrid Systems : Computation and Control (HSCC'05)*, volume 3414 of *Lecture Notes in Computer Science*, pages 258–273. Springer, 2005.

[GAO92] GAO. Patriot missile defense : Software problem led to system failure at Dhahran, Saudi Arabia. Technical report, U.S. General Accouting Office, 1992.

[GG08] C. Le Guernic and A. Girard. Zonotope-hyperplane intersection for hybrid systems reachability analysis. In *Proceedings of the International Workshop on Hybrid Systems : Computation and Control (HSCC'08)*, volume 4981 of *Lecture Notes in Computer Science*, pages 215–228. Springer, 2008.

[GHK+03] G. Gierz, K.H. Hofmann, K. Keimel, J.D. Lawson, M. Mislove, and D.S. Scott. *Continuous Lattices and Domains*. Cambridge University Press, 2003.

[GJS98] V. Gupta, R. Jagadeesan, and V. Saraswat. Computing with continuous change. *Science of Computer Programming*, 30(1–2) :3–49, January 1998.

[GJSB95] V. Gupta, R. Jagadeesan, V. Saraswat, and D. Bobrow. Programming in hybrid constraint languages. In *Hybrid Systems II*, volume 999 of *Lecture Notes in Computer Science*. Springer Verlag, 1995.

[GMP02] E. Goubault, M. Martel, and S. Putot. Asserting the precision of floating-point computations : A simple abstract interpreter. In *Proceedings of the 11th European Symposium on Programming Languages and Systems (ESOP'02)*, volume 2305 of *Lecture Notes in Computer Science*, pages 209–212. Springer, 2002.

[GMP06] E. Goubault, M. Martel, and S. Putot. Some future challenges in the validation of control systems. In *European Congress on Embedded Real-Time Software (ERTS)*. SEE, 2006.

[Gou01] E. Goubault. Static analyses of floating-point operations. In *Proceedings of the 8th International Static Analysis Symposium (SAS'01)*, volume 2126 of *Lecture Notes in Computer Science*, pages 234–259. Springer, 2001.

[GP06] E. Goubault and S. Putot. Static analysis of numerical algorithms. In *Proceedings of the 13th International Symposium on Static Analysis (SAS'06)*, volume 4134 of *Lecture Notes in Computer Science*, pages 18–34. Springer, 2006.

[Gra91] P. Granger. Static analysis of linear congruence equalities among variables of a program. In *International joint conference on theory and practice of software development on Colloquium on trees in algebra and programming (TAPSOFT'91)*, volume 493 of *Lecture Notes in Computer Science*, pages 169–192. Springer, 1991.

[Gri00] A. Griewank. *Evaluating derivatives : principles and techniques of algorithmic differentiation*. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 2000.

[GSS95] V. Gupta, V. Saraswat, and P. Struss. A model of a photocopier paper path. In *Workshop on Engineering Problems for Qualitative Reasoning*, August 1995.

[Gus91] K. Gustafsson. Control theoretic techniques for stepsize selection in explicit runge-kutta methods. *ACM Transaction on Mathematical Software*, 17(4) :533–554, 1991.

[Gus93] K. Gustafsson. *Control of error and convergence in ODE solvers*. PhD thesis, Lund Institute of Technology, 1993.

[HDB01] W. Heemels, B. De Schutter, and A. Bemporad. Equivalence of hybrid dynamical models. *Automatica*, 37(7) :1085–1091, July 2001.

[Hen96] T. A. Henzinger. The theory of hybrid automata. In *Symposium on Logic in Computer Science*, pages 278–292. IEEE Computer Society Press, 1996.

[Hes04] J. Hespanha. Uniform stability of switched linear systems : Extensions of LaSalle's invariance principle. 49(4) :470–482, 2004.

[HF05] R. Heckmann and C. Ferdinand. Verifying safety-critical timing and memory-usage properties of embedded software by abstract interpretation. In *Proceedings of the conference on Design, Automation and Test in Europe (DATE'05)*, pages 618–619. IEEE Computer Society, 2005.

[HH95] T. A. Henzinger and P. Ho. A note on abstract interpretation strategies for hybrid automata. In *Hybrid Systems II*, volume 999 of *Lecture Notes in Computer Science*, pages 252–264. Springer-Verlag, 1995.

[HHMWT00] T. A. Henzinger, B. Horowitz, R. Majumdar, and H. Wong-Toi. Beyond HYTECH : Hybrid systems analysis using interval numerical methods. In *Third International Workshop on Hybrid Systems : Computation and Control (HSCC'00)*, volume 1790 of *Lecture Notes in Computer Science*, pages 130–144. Springer, 2000.

[HHWT97] T. A. Henzinger, P. H. Ho, and H. Wong-Toi. HyTech : A model checker for hybrid systems. In *Proceedings of the 9th International Conference Computer Aided Verification*, volume 1254 of *Lecture Notes in Computer Science*, pages 460–463. Springer, 1997.

[HHWT98] T. A. Henzinger, P. H. Ho, and H. Wong-Toi. Algorithmic analysis of nonlinear hybrid systems. *IEEE Transactions on Automatic Control*, 43 :540–554, 1998.

[HKPV95] T. A. Henzinger, P. W. Kopke, A. Puri, and P. Varaiya. What's decidable about hybrid automata ? In *ACM Symposium on Theory of computing*, pages 373–382. ACM, 1995.

[HL08] C. Hymans and O. Levillain. Newspeak, Doubleplussimple Minilang for Goodthinkful Static Analysis of C. Technical Note 2008-IW-SE-00010-1, EADS IW/SE, 2008.

[HLMR05] M. Heymann, F. Lin, G. Meyer, and S. Resmerita. Analysis of zeno behaviors in hybrid systems. *IEEE Transactions on Automatic Control*, 50(3) :376–384, 2005.

[HM00] T. Henzinger and R. Majumdar. Symbolic model checking for rectangular hybrid systems. In *Proceedings of the Sixth International Workshop on Tools and Algorithms for the Construction and Analysis of Systems*, pages 142–156. Lecture Notes in Computer Science 1785, Springer-Verlag, January 2000.

[Hoa78] C. A. R. Hoare. Communicating sequential processes. *Commun. ACM*, 21(8) :666–677, 1978.

[Hoa85] C. A. R. Hoare. *Communicating Sequential Processes*. Prentice-Hall, 1985.

[HPR94] N. Halbwachs, Y. Proy, and P. Raymond. Verification of linear hybrid systems by means of convex approximations. In *First International Static Analysis Symposium (SAS'94)*, volume 864 of *Lecture Notes in Computer Science*, pages 223–237. Springer, 1994.

[HR98] T. A. Henzinger and V. Rusu. Reachability verification for hybrid automata. In *First InternationalWorkshop on Hybrid Systems : Computation and Control (HSCC'98)*,

volume 1386 of *Lecture Notes in Computer Science*, pages 190–204. Springer-Verlag, 1998.

[Inc56] E. Ince. *Ordinary Differential Equations*. Dover Publications, 1956.

[JELS99] K.H. Johansson, M. Egerstedt, J. Lygeros, and S. Sastry. On the regularization of zeno hybrid automata. *Systems and control letters*, 38(3) :141–150, 1999.

[JKDW01] L. Jaulin, M. Kieffer, O. Didrit, and E. Walter. *Applied Interval Analysis*. Springer, 2001.

[JLSE99] K. Johansson, J. Lygeros, S. S. Sastry, and M. Egerstedt. Simulation of zeno hybrid automata. In *IEEE Conference on Decision and Control*, pages 3538–3543, 1999.

[Joh77] S. Johnson. Lint, a C program checker. Technical Report 65, Bell Labs, 1977.

[Knu94] O. Knuppel. PROFIL/BIAS : a fast interval library. *Computing*, 53(3-4) :277–287, 1994.

[Koz83] Dexter Kozen. Results on the propositional [mu]-calculus. *Theoretical Computer Science*, 27(3) :333–354, 1983.

[KSF+99] S. Kowalewski, O. Stursberg, M. Fritz, H. Graf, I. H., J. Preuß, and et al. A case study in tool-aided analysis of discretely controlled continuous systems : the two tanks problem. In *Hybrid Systems V*, volume 1567 of *Lecture Notes in Computer Science*. Springer, 1999.

[LJS+03] J. Lygeros, K.H. Johansson, S.N. Simic, J. Zhang, and S.S. Sastry. Dynamical properties of hybrid automata. *IEEE Transactions on Automatic Control*, 48(1) :2–17, 2003.

[Lor63] E.N. Lorenz. Deterministic nonperiodic flow. *Journal of the Atmospheric Sciences*, 20(2) :130–141, 1963.

[LS07] Y. Lin and M. A. Stadtherr. Validated solutions of initial value problems for parametric ODEs. *Applied Numererical Mathematics*, 57(10) :1145–1162, 2007.

[LSVW96] N. Lynch, R. Segala, F. Vaandrager, and H.B. Weinberg. Hybrid I/O automata. In *Proceedings of the DIMACS/SYCON workshop on Hybrid systems III : verification and control*, pages 496–510. Springer, 1996.

[Lö88] R. Löhner. *Einschließung der Lösung gewöhnlicher Anfangsund Randwertaufgaben und Anwendungen*. PhD thesis, Universität Karlsruhe, 1988.

[Lö95] R.J. Löhner. Step size and order control in the verified solution of IVP with ODE's. In *International conference on scientific computation and differential equations (SciSCADE'95)*, 1995.

[Mar02] M. Martel. Propagation of roundoff errors in finite precision computations : A semantics approach. In *Proceedings of the 11th European Symposium on Programming Languages and Systems (ESOP'02)*, volume 2305 of *Lecture Notes in Computer Science*, pages 194–208. Springer, 2002.

[Mar05] M. Martel. An overview of semantics for the validation of numerical programs. In *Proceedings of the 6th International Conference on Verification, Model Checking and Abstract Interpretation (VMCAI'05)*, volume 3385 of *Lecture Notes in Computer Science*, pages 59–77. Springer, 2005.

[Mar06] M. Martel. Semantics of roundoff error propagation in finite precision calculations. *Higher Order Symbolic Computation*, 19(1) :7–30, 2006.

[Mas02] D. Massé. *Temporal Property-driven Verification by Abstract Interpretation*. PhD thesis, École Polytechnique, 2002.

[Mei98] J. Meinadier. *Ingénierie et intégration des systèmes*. Collection Etudes et logiciels informatiques. Hermès - Lavoisier, 1998.

[Mel94] P. Mellor. CAD : Computer aided disaster. In *SOFSEM*, pages 147–180, 1994.

[Mil99] R. Milner. *Communicating and Mobile Systems : the Pi-Calculus*. Cambridge University Press, June 1999.

[Mil00] J.S. Miller. Decidability and complexity results for timed automata and semi-linear hybrid automata. In *Third International Workshop on Hybrid Systems : Computation and Control (HSCC'00)*, volume 1790 of *Lecture Notes in Computer Science*, pages 296–309. Springer-Verlag, 2000.

[Min01] A. Miné. The octagon abstract domain. In *Proceedings of the Eighth Working Conference on Reverse Engineering (WCRE'01)*, IEEE, pages 310–319. IEEE Computer Society, October 2001.

[MK06] I. Ben Makhlouf and S. Kowalewski. An evaluation of two recent reachability analysis tools for hybrid systems. In *Second IFAC Conference on Analysis and Design of Hybrid Systems*, pages 377–382. ELSEVIER, 2006.

[Moo66] R. Moore. *Interval Analysis*. Prentice-Hall, 1966.

[Moo79] R. Moore. *Methods and Applications of Interval Analysis*. SIAM, Philadelphia, PA, 1979.

[Ned06] N.S. Nedialkov. Interval tools for ODEs and DAEs. Technical Report CAS 06-09-NN, Dept. of Computing and Software, McMaster University, 2006.

[NGC08] M. Nita, D. Grossman, and C. Chambers. A theory of platform-dependent low-level software. In *Proceedings of the 35th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'08)*, pages 209–220. ACM, 2008.

[NJ99] N. S. Nedialkov and K. R. Jackson. An interval Hermite-Obreschkoff method for computing rigorous bounds on the solution of an initial value problem for an ordinary differential equation. In *Developments in Reliable Computing*, pages 289–310. Kluwer, Dordrecht, Netherlands, 1999.

[NJ01a] N. S. Nedialkov and K. R. Jackson. The design and implementation of an object-oriented validated ODE solver, 2001.

[NJ01b] N.S. Nedialkov and K.R. Jackson. A new perspective on the wrapping effect in interval methods for initial value problems for ordinary differential equations. *Perspectives on Enclosure Methods*, pages 219–264, 2001.

[NJC99] N.S. Nedialkov, K.R. Jackson, and G.F. Corliss. Validated solutions of initial value problems for ordinary differential equations. *Applied Mathematics and Computation*, 105(1) :21–68, 1999.

[NJP01] N.S. Nedialkov, K.R. Jackson, and J.D. Pryce. An effective high-order interval method for validating existence and uniqueness of the solution of an IVP for an ODE. *Reliable Computing*, 7 :449–465(17), December 2001.

[OEM99] M. Otter, H. Elmqvist, and S.E. Mattsson. Hybrid modeling in modelica based on the synchronous data flow principle. In *IEEE Symposium on Computer-Aided Control System Design, CACSD'99*, pages 151–157, 1999.

[P7585] IEEE Task P754. *ANSI/IEEE 754-1985, Standard for Binary Floating-Point Arithmetic*. IEEE, New York, August 12 1985.

[PC07] A. Platzer and E. Clarke. The image computation problem in hybrid systems model checking. In *10th International Workshop on Hybrid Systems : Computation and Control (HSCC'07)*, volume 4416 of *Lecture Notes in Computer Science*, pages 473–486. Springer, 2007.

[PGM03] S. Putot, E. Goubault, and M. Martel. Static analysis-based validation of floating-point computations. In *Numerical Software with Result Verification*, volume 2991 of *Lecture Notes in Computer Science*, pages 306–313. Springer, 2003.

[Pop06] S. Pop. *The SSA Representation Framework : Semantics, Analyses and GCC Implementation*. PhD thesis, Ecole des Mines de Paris, December 2006.

[Ral81] L.B. Rall. *Automatic Differentiation : Techniques and Applications*, volume 120 of *Lecture Notes in Computer Science*. Springer, 1981.

[RMB05] N. Revol, K. Makino, and M. Berz. Taylor models and floating-point arithmetic : proof that arithmetic operations are validated in COSY. *Journal of Logic and Algebraic Programming*, 64 :135–154, 2005.

[RS02] W.C. Rounds and H. Song. The phi-calculus - a hybrid extension of the pi-calculus to embedded systems. Technical report, University of Michigan, 2002.

[RS03] W. C. Rounds and H. Song. The phi-calculus : A language for distributed control of reconfigurable embedded systems. In *6th International Workshop on Hybrid Systems : Computation and Control (HSCC'03)*, volume 2623 of *Lecture Notes in Computer Science*, pages 435–449. Springer, 2003.

[SB93] J. Stoer and R. Bulirsch. *Introduction to Numerical Analysis*. Springer-Verlag, 1993.

[Sha05] L. F. Shampine. Error estimation and control for odes. *Journal of Scientific Computing*, 25(1-2) :3–16, 2005.

[SRP91] V. Saraswat, M. Rinard, and P. Panangaden. Semantic foundations of concurrent constraint programming. In *Conference Record of the 28th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'01)*, pages 333–352. ACM, 1991.

[SS00] A.J. van der Schaft and J.M. Schumacher. *An Introduction to Hybrid Dynamical Systems*, volume 251 of *Lecture Notes in Control and Information Sciences*. Springer, London, 2000.

[Sta97] Ole Stauning. *Automatic Validation of Numerical Solutions*. PhD thesis, Technical University of Denmark, Lyngby, Denmark, 1997.

[Sto96] N. Storey. *Safety Critical Computer Systems*. Addison-Wesley Longman Publishing Co., Inc, 1996.

[Str99] K. Strehl. Interval diagrams : Increasing efficiency of symbolic real-time verification. In *Proceedings of the Sixth International Conference on Real-Time Computing Systems and Applications (RTCSA'99)*, page 488. IEEE Computer Society, 1999.

[Tar55] A. Tarski. A lattice-theoretical fixpoint theorem and its applications. *Pacific Journal of Mathematics*, 5 :285–309, 1955.

[VB04] A. Venet and G. Brat. Precise and efficient static array bound checking for large embedded c programs. *SIGPLAN Notices*, 39(6) :231–242, 2004.

[vBMR+06] D. van Beek, K. Man, M. Reniers, J. Rooda, and R. Schiffelers. Syntax and consistent equation semantics of hybrid-chi. *Journal of Logic and Algebraic Programming*, 68(1-2) :129–210, 2006.

[VJ02] D. Vandevoorde and N.M. Josuttis. *C++ Templates*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2002.

[Win93] G. Winskel. *The formal semantics of programming languages : an introduction*. MIT Press, Cambridge, MA, USA, 1993.

[ZJLS01] J. Zhang, K. Johansson, J. Lygeros, and S. Sastry. Zeno hybrid systems. *International Journal of Robust and Nonlinear Control*, 11 :435–451, 2001.

ANNEXE A

Simulation du système des deux réservoirs en Simulink

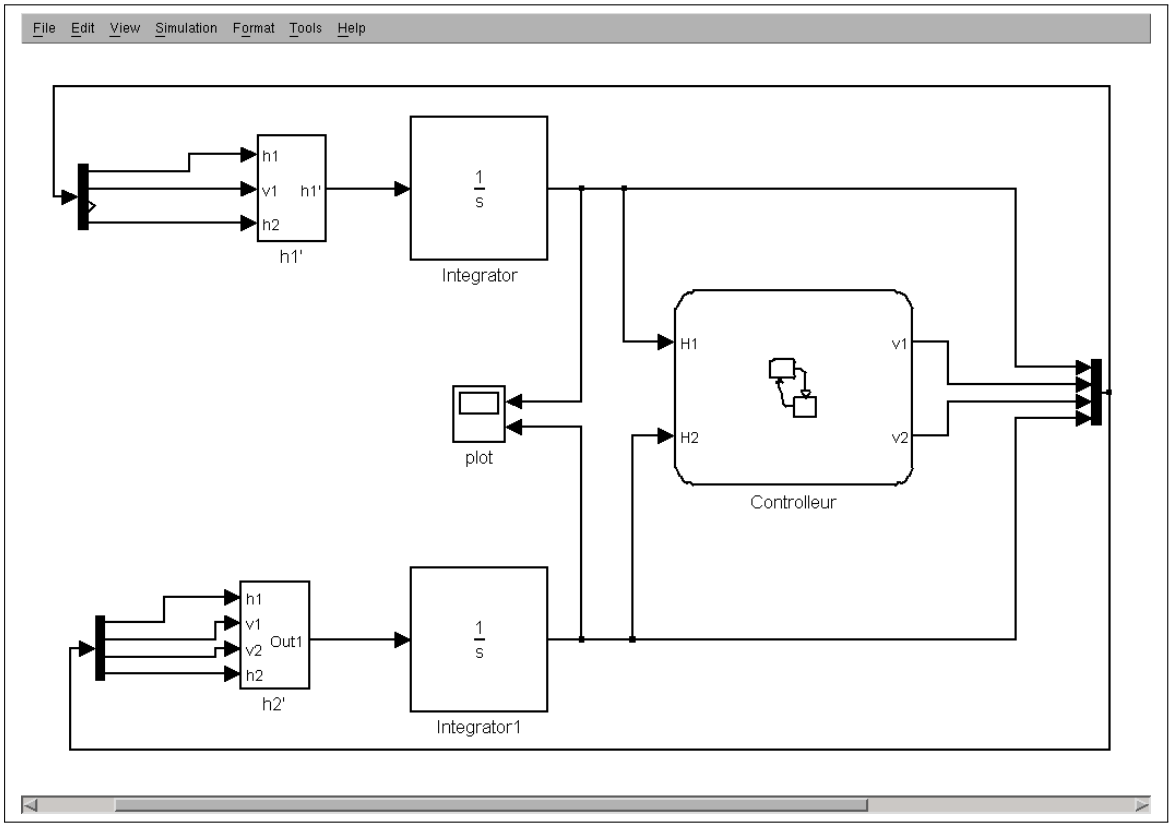


FIG. A.1 – Programme principale.

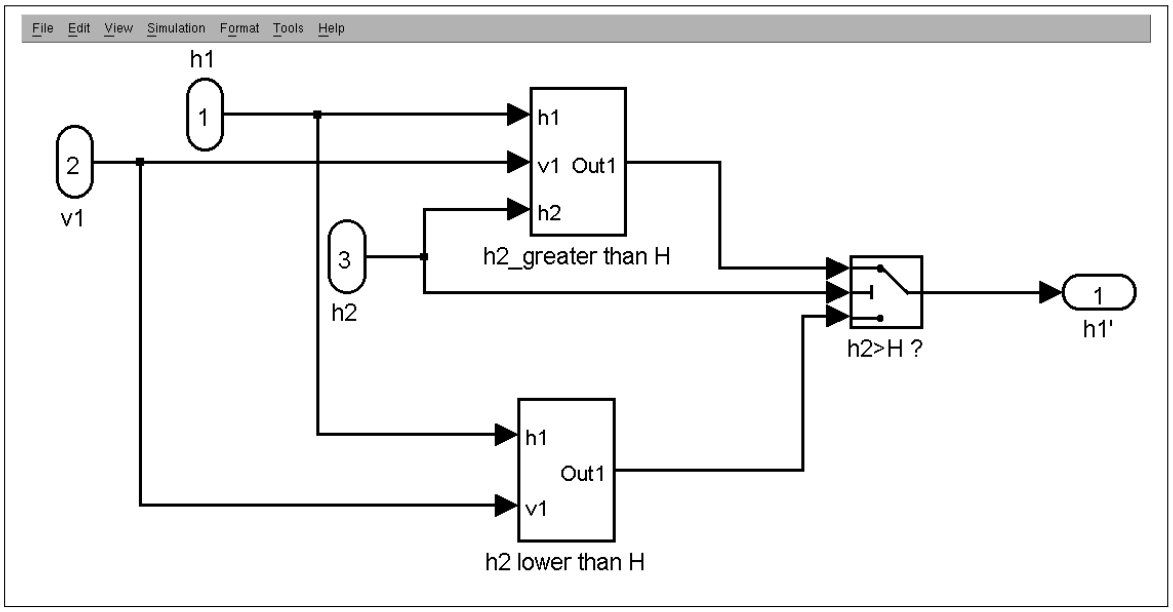


FIG. A.2 – Équation différentielle pour h_1.

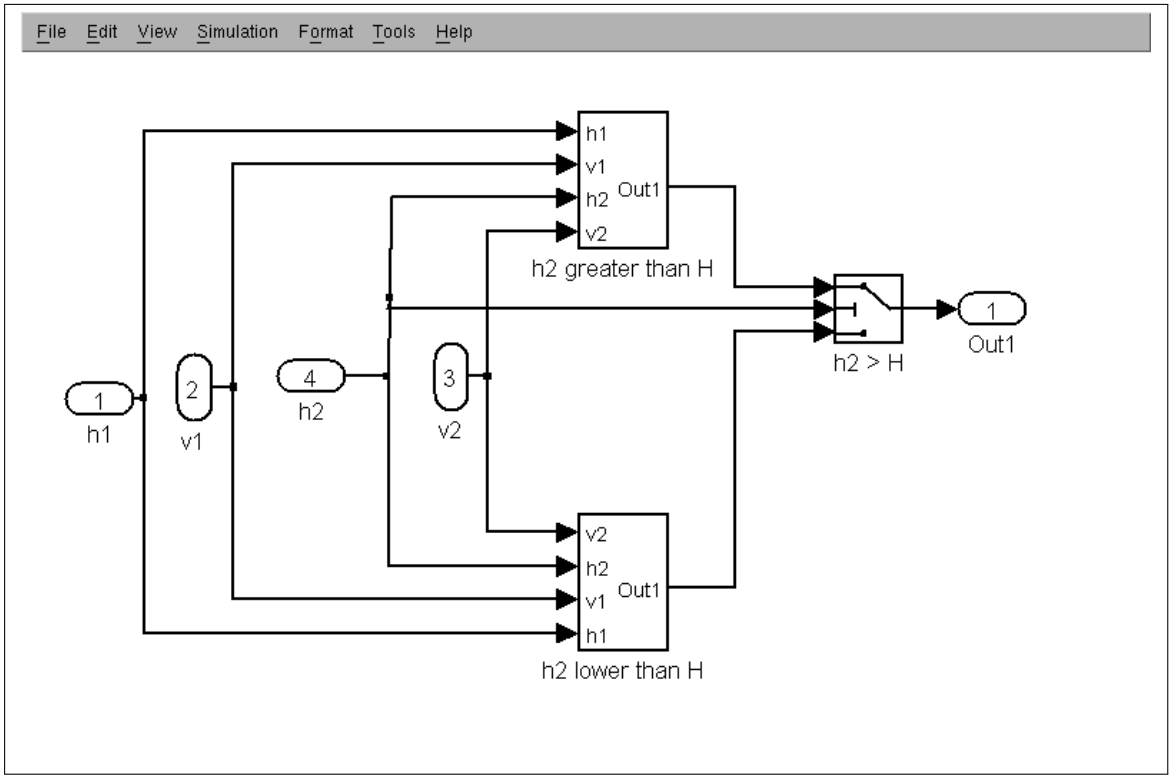


FIG. A.3 – Équation différentielle pour h_2.

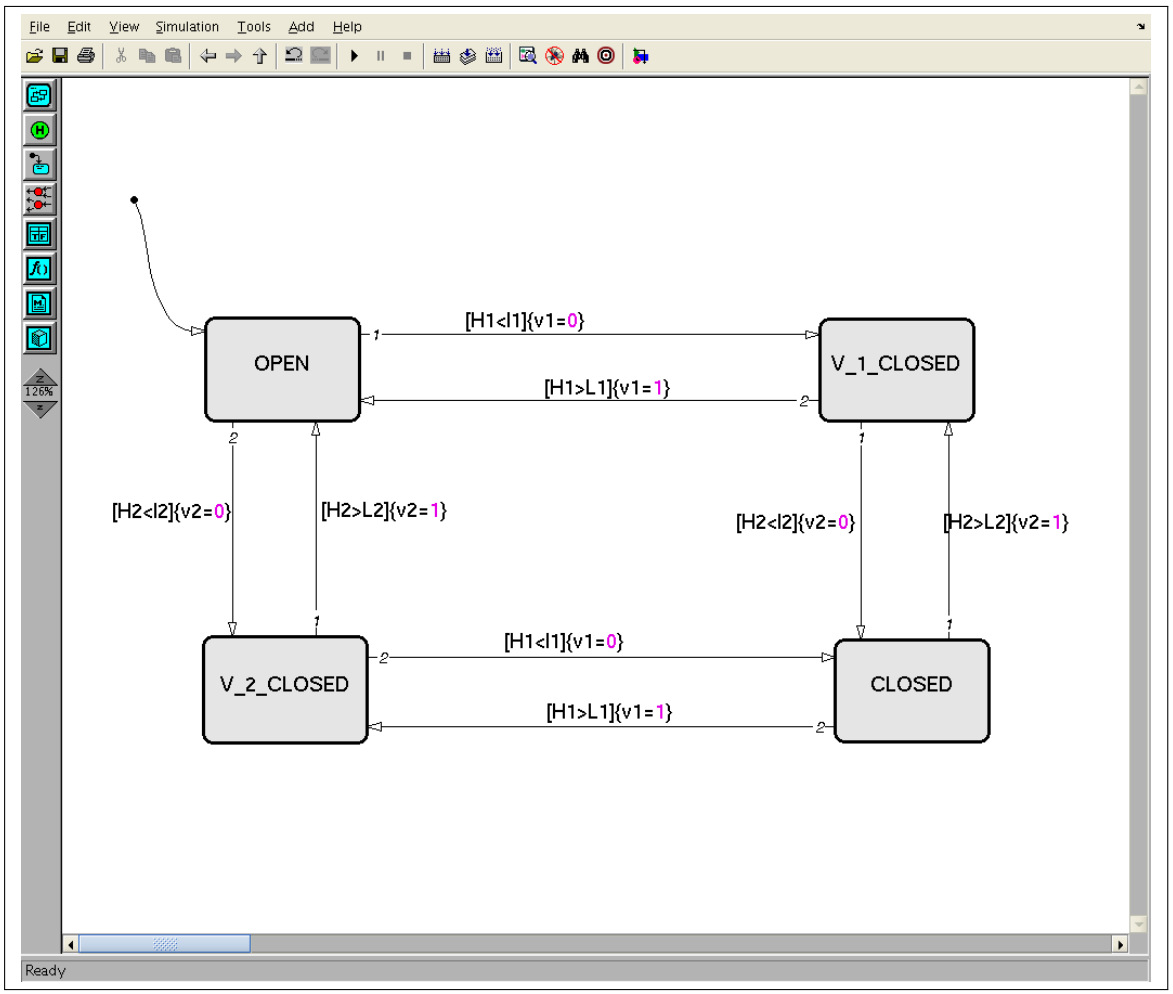


FIG. A.4 – Stateflow représentant le controlleur.

ANNEXE B

Preuves de correction des encadrements de GRKLib

B.1 Preuve de la proposition 6.13

Soit un problème de Cauchy de dimension 1 $\dot{y} = F(y, x),\ y(x_0) = y_0$, où F est une fonction 5 fois dérivable. Nous nous limitons au cas de la dimension 1 car la preuve en dimension n quelconque est en tout point similaire, mais nettement plus pénible à écrire et lire, car nous devrions alors calculer toutes les dérivées du style $\frac{\partial^k F}{\partial^{k_1} y_1 \partial^{k_2} y_2 ... \partial^{k_n} y_n}$. Notons y_∞ la solution du problème de Cauchy et soit $\tilde{y}$ la fonction définie par :

$$\begin{aligned}
\tilde{y}(x) &= y_0 + \frac{x - x_0}{2}\left(k_1(x) + 2k_2(x) + 2k_3(x) + k_4(x)\right) \\
k_1(x) &= F\left(y_0, x_0\right) \\
k_2(x) &= F\left(y_0 + \frac{x - x_0}{2} k_1(x), x_0 + \frac{x - x_0}{2}\right) \\
k_3(x) &= F\left(y_0 + \frac{x - x_0}{2} k_2(x), x_0 + \frac{x - x_0}{2}\right) \\
k_4(x) &= F\left(y_0 + (x - x_0)k_3(x), x_0 + (x - x_0)\right)
\end{aligned}$$

Nous devons démontrer la proposition suivante.

Proposition B.1 *Les quatre premières dérivées de y_∞ et $\tilde{y}$ sont égales en $x = x_0$:*

$$\forall i \in [0, 4],\ \frac{d^i y}{dx^i}(x_0) = \frac{d^i \tilde{y}}{dx^i}(x_0).$$

Preuve Soit g la fonction définie par $\forall x,\ g(x) = y(x) - \tilde{y}(x)$. Nous devons donc montrer que :

$$\forall i \in [0, 4],\ \frac{d^i g}{dx^i}(x_0) = 0\ .$$

Pour $i = 0$, on a $g(x_0) = y(x_0) = \tilde{y}(x_0) = y_0 - y_0 = 0$. On s'attaque donc au cas $i \in [1, 4]$.

Calculons dans un premier temps $\frac{d^i F}{dx^i}$ pour tout $i \in [1,4]$:

$$\begin{aligned}
\frac{dy}{dx} &= f(y,x) \\
\frac{d^2y}{dx^2} &= \frac{df}{dx} = \frac{\partial f}{\partial x} + \frac{dy}{dx}\frac{\partial f}{\partial y} = \frac{\partial f}{\partial x} + f.\frac{\partial f}{\partial y} \\
\frac{d^3y}{dx^3} &= \frac{d}{dx}\left(\frac{\partial f}{\partial x} + f.\frac{\partial f}{\partial y}\right) = \frac{\partial^2 f}{\partial x^2} + 2f\frac{\partial^2 f}{\partial x \partial y} + f^2\frac{\partial^2 f}{\partial y^2} + \frac{\partial f}{\partial x}.\frac{\partial f}{\partial y} + f.\left(\frac{\partial f}{\partial y}\right)^2 \\
\frac{d^4y}{dx^4} &= \frac{d}{dx}\left(\frac{d^3y}{dx^3}\right) \\
&= \frac{\partial^3 f}{\partial x^3} + 3f\frac{\partial^3 f}{\partial x^2 \partial y} + 3f^2\frac{\partial^3 f}{\partial x \partial y^2} + f^3\frac{\partial^3 f}{\partial y^3} \\
&\quad + 3\frac{\partial^2 f}{\partial x \partial y}.\frac{df}{dx} + 3f\frac{df}{dx}.\frac{\partial^2 f}{\partial y^2} + \frac{d^2 f}{dx^2}.\frac{\partial f}{\partial y}
\end{aligned}$$

On calcule maintenant $\frac{d^i \tilde{y}}{dx^i}(x_0)$ pour $i \in [1,4]$. Pour cela, nous allons calculer, pour tout $j \in [1,4]$, $\frac{d^i\left((x-x_0)k_j\right)}{dx^i}(x_0)$. Clairement :

$$\forall j \in [1,4],\ \forall i \in [1,4],\ \frac{d(x-x_0)k_j}{dx^i} = (x-x_0)\frac{d^i k_j}{dx^i} + \frac{d^{(i-1)}k_j}{dx^{(i-1)}} \tag{B.1}$$

et donc :

$$\forall j \in [1,4],\ \forall i \in [1,4],\ \frac{d^i(x-x_0)k_j}{dx^i}(x_0) = i.\frac{d^{(i-1)}k_j}{dx^{(i-1)}}(x_0)\ . \tag{B.2}$$

Pour $i = 1$, on a donc :

$$\frac{d\tilde{y}}{dx}(x_0) = \frac{1}{6}\left(k_1(x_0) + 2k_2(x_0) + 2k_3(x_0) + k_4(x_0)\right) = \frac{1}{6}.6f(y_0,x_0) = f(y_0,x_0) = \frac{dy}{dx}(x_0)$$

et donc $\frac{dg}{dx}(x_0) = 0$. Pour les dérivées d'ordre supérieures, nous devons d'abord calculer les terms $\frac{d^i k_j}{dx^i}(x_0)$ pour $i \in [2,4]$ et $j \in [1,4]$. Comme k_1 est constant, on a clairement $\forall i \in [2,4],\ \frac{d^i k_1}{dx^i} = 0$. On montre maintenant par récurrence que :

$$\forall i \geq 1,\ \ \frac{d^i k_2}{dx^i} = \frac{1}{2^i}\left(\sum_{l=0}^{i}\left(\begin{array}{c} i \\ l \end{array}\right)(k_1)^l\frac{\partial^i f}{\partial x^{i-l}\partial y^l}\right) \tag{B.3}$$

où les dérivées sont évaluées en $\left(y_0 + \frac{x-x_0}{2}k_1(x), x_0 + \frac{(x-x_0)}{2}\right)$. Si $i = 1$, on a :

$$\frac{dk_2}{dx} = \frac{d}{dx}\left(x_0 + \frac{x-x_0}{2}\right)\frac{\partial f}{\partial x} + \frac{d}{dx}\left(y_0 + \frac{x-x_0}{2}k_1\right)\frac{\partial f}{\partial y} = \frac{1}{2}\left(\frac{\partial f}{\partial x} + k_1\frac{\partial f}{\partial y}\right)\ .$$

Soit maintenant $i \geq 1$ tel que l'équation (B.3) est vraie. Alors, on a :

$$\begin{aligned}
\frac{d^{(i+1)}k_2}{dx^{(i+1)}} &= \frac{d}{dx}\left(\frac{1}{2^i}\left(\sum_{l=0}^{i}\left(\begin{array}{c} i \\ l \end{array}\right)(k_1)^l\frac{\partial^i f}{\partial x^{i-l}\partial y^l}\right)\right) \\
&= \frac{1}{2^i}\left(\sum_{l=0}^{i}\left(\begin{array}{c} i \\ l \end{array}\right)(k_1)^l\frac{d}{dx}\left(\frac{\partial^i f}{\partial x^{i-l}\partial y^l}\right)\right) \\
&= \frac{1}{2^i}\left(\sum_{l=0}^{i}\left(\begin{array}{c} i \\ l \end{array}\right)\frac{(k_1)^l}{2}\frac{\partial^{(i+1)}f}{\partial x^{(i+1-l)}\partial y^l}\right) + \frac{1}{2^i}\left(\sum_{l=0}^{i}\left(\begin{array}{c} i \\ l \end{array}\right)\frac{(k_1)^{l+1}}{2}\frac{\partial^{(i+1)}f}{\partial x^{(i-l)}\partial y^{l+1}}\right) \\
&= \frac{1}{2^{i+1}}\left(\sum_{l=0}^{i}\left(\begin{array}{c} i \\ l \end{array}\right)(k_1)^l\frac{\partial^{(i+1)}f}{\partial x^{(i+1-l)}\partial y^l}\right) + \frac{1}{2^{i+1}}\left(\sum_{l=1}^{i+1}\left(\begin{array}{c} i \\ l-1 \end{array}\right)(k_1)^l\frac{\partial^{(i+1)}f}{\partial x^{(i+1-l)}\partial y^l}\right) \\
&= \frac{1}{2^{i+1}}\left(\sum_{l=1}^{i}\left(\left(\begin{array}{c} i \\ l \end{array}\right) + \left(\begin{array}{c} i \\ l-1 \end{array}\right)\right)(k_1)^l\frac{\partial^{(i+1)}f}{\partial x^{(i+1-l)}\partial y^l}\right) + \frac{1}{2^{i+1}}\frac{\partial^{i+1}f}{\partial x^{i+1}} + \frac{1}{2^{i+1}}\left(\begin{array}{c} i \\ i \end{array}\right)(k_1)^{i+1}\frac{\partial^{i+1}f}{\partial y^{i+1}} \\
&= \frac{1}{2^{i+1}}\left(\sum_{l=0}^{i+1}\left(\begin{array}{c} i+1 \\ l \end{array}\right)(k_1)^l\frac{d}{dx}\left(\frac{\partial^{i+1}f}{\partial x^{i+1-l}\partial y^l}\right)\right)
\end{aligned}$$

ce qui prouve la récurrence. L'équation (B.3) est donc vraie pour tout $i \in \mathbb{N}$.
Calculons maintenant les dérivées de k_3. Soit $k_2^1(x) = \frac{d((x-x_0)k_2)}{dx}$, $k_2^2(x) = \frac{d^2((x-x_0)k_2)}{dx^2}$, $k_2^3(x) =$

$\frac{d^3((x-x_0)k_2(x))}{dx^3}$ et $k_2^4(x) = \frac{d^4((x-x_0)k_2(x))}{dx^4}$. Soit $y^* = y_0 + \frac{x-x_0}{2}k_2(x), x^* = x_0 + \frac{x-x_0}{2}$. On montre par récurrence que, si toutes les dérivées sont calculées en (y^*, x^*), on a :

$$\forall i \geq 1,\ \frac{d^i k_3}{dx^i} = \sum_{k=1}^{i}\left(\sum_{l=0}^{k}\alpha_{k,l}^i \frac{\partial^k f}{\partial x^{k-l}\partial y^l}\right) \tag{B.4}$$

où pour tout k, l, i, $\alpha_{k,l}^i$ est un polynôme en k_2^1, k_2^2, k_2^3 and k_2^4 tel que :

- $\forall i,\ k \leq 0\ \Rightarrow \alpha_{k,l}^i = 0$ et $(l \leq 0$ ou $l \geq k) \Rightarrow \alpha_{k,l}^i = 0$
- $\forall l \in [0,i],\ \alpha_{i,l}^i = \frac{1}{2^i}\begin{pmatrix} i \\ l \end{pmatrix}\left(k_2^1\right)^l$
- $\forall i,\ \ \forall i \geq k \geq l \geq 1,\ \alpha_{k,l}^{i+1} = \frac{d\alpha_{k,l}^i}{dx} + \frac{1}{2}\left(\alpha_{k-1,l}^i + \alpha_{k-1,l-1}^i k_2^1\right)$

Notons qu'en particulier, on a : $\forall i,\ \alpha_{1,1}^i = \frac{1}{2}k_2^i$. Prouvons maintenant par récurrence l'équation (B.4). Si $i = 1$, on a :

$$\frac{dk_3}{dx} = \frac{1}{2}\left(\frac{\partial f}{\partial x} + k_2^1(x).\frac{\partial f}{\partial y}\right)\ .$$

Soit maintenant $i \geq 1$ tel que l'équation (B.4) est vraie. Alors, on a :

$$\begin{aligned}
\frac{d^{(i+1)}k_3}{dx} &= \frac{d}{dx}\left(\frac{1}{2^i}\left(\sum_{l=0}^{i}\begin{pmatrix} i \\ l \end{pmatrix}\left(k_2^1\right)^l \frac{\partial^i f}{\partial x^{i-l}\partial y^l}\right) + \sum_{k=1}^{i-1}\left(\sum_{l=1}^{k}\alpha_{k,l}^i \frac{\partial^k f}{\partial x^{k-l}\partial y^l}\right)\right) \\
&= \frac{1}{2^{i+1}}\left(\sum_{l=0}^{i+1}\begin{pmatrix} i+1 \\ l \end{pmatrix}\left(k_2^1\right)^l \frac{\partial^{i+1} f}{\partial x^{i+1-l}\partial y^l}\right) + \frac{1}{2^i}\left(\sum_{l=0}^{i} l\binom{i}{l}k_2^2\left(k_2^1\right)^{(l-1)}\frac{\partial^i f}{\partial x^{i-l}\partial y^l}\right) \\
&\quad + \sum_{k=1}^{i-1}\sum_{l=1}^{k}\left(\frac{d\alpha_{k,l}^i}{dx}\frac{\partial^k f}{\partial x^{k-l}\partial y^l} + \frac{\alpha_{k,l}^i}{2}\frac{\partial^{k+1} f}{\partial x^{k+1-l}\partial y^l} + \frac{\alpha_{k,l}^i k_2^1}{2}\frac{\partial^{k+1} f}{\partial x^{k-l}\partial y^{l+1}}\right) \\
&= \frac{1}{2^{i+1}}\left(\sum_{l=0}^{i+1}\begin{pmatrix} i+1 \\ l \end{pmatrix}\left(k_2^1\right)^l \frac{\partial^{i+1} f}{\partial x^{i+1-l}\partial y^l}\right) + \sum_{k=1}^{i}\sum_{l=1}^{k}\frac{d\alpha_{k,l}^i}{dx}\frac{\partial^k f}{\partial x^{k-l}\partial y^l} \\
&\quad + \frac{1}{2}\sum_{k=1}^{i-1}\sum_{l=1}^{k+1}\left(\alpha_{k,l} + \alpha_{k,l-1}k_2^1\right)\frac{\partial^{k+1} f}{\partial x^{k+1-l}\partial y^l}
\end{aligned}$$

En réarrangeant les termes de la dernière somme, on obtient :

$$\begin{aligned}
\frac{d^{(i+1)}k_3}{dx} &= \frac{1}{2^{i+1}}\left(\sum_{l=0}^{i+1}\begin{pmatrix} i+1 \\ l \end{pmatrix}\left(k_2^1\right)^l \frac{\partial^{i+1} f}{\partial x^{i+1-l}\partial y^l}\right) + \sum_{k=2}^{i}\sum_{l=1}^{k}\left(\frac{d\alpha_{k,l}^i}{dx} + \frac{1}{2}\left(\alpha_{k-1,l}^i + \alpha_{k-1,l-1}^i k_2^1\right)\right)\frac{\partial^k f}{\partial x^{k-l}\partial y^l} \\
&\quad + \frac{d\alpha_{1,1}^i}{dx}\frac{\partial f}{\partial y} \\
&= \sum_{k=1}^{i+1}\left(\sum_{l=1}^{k}\alpha_{k,l}^{i+1}\frac{\partial^k f}{\partial x^{k-l}\partial y^l}\right)
\end{aligned}$$

ce qui termine la récurrence.
On calcule enfin les dérivées de k_4. Soit $y^* = y_0 + \frac{x-x_0}{2}k_3(x)$ et $k_3^1(x) = \frac{d((x-x_0)k_3)}{dx}$, $k_3^2(x) = \frac{d^2((x-x_0)k_3)}{dx^2}$, $k_3^3(x) = \frac{d^3((x-x_0)k_3(x))}{dx^3}$, $k_3^4(x) = \frac{d^4((x-x_0)k_3(x))}{dx^4}$. On prouve alors par une récurrence identique à la précédente que :

$$\forall i,\ \frac{d^i k_4}{dx^i} = \sum_{k=1}^{i}\left(\sum_{l=1}^{k}\beta_{k,l}^i \frac{\partial^k f}{\partial x^{k-l}\partial y^l}\right)$$

où toutes les dérivées sont évaluées en (y^*, x^*) et :

- $\forall i,\ k \leq 0 \Rightarrow \beta_{k,l}^i = 0$ et $\ (l \leq 0$ ou $l \geq k) \Rightarrow \beta_{k,l}^i = 0$
- $\forall i,\ \forall l \in [1,i],\ \beta_{i,l}^i = \begin{pmatrix} i \\ l \end{pmatrix}\left(k_3^1\right)^l$
- $\forall i,\ \ \forall i \geq k \geq l \geq 1,\ \beta_{k,l}^{i+1} = \frac{d\beta k,l}{dx} + \beta_{k-1,l}^i + \beta_{k-1,l-1}^i k_3^1$

On peut donc maintenant calculer les premiers termes des suites $\left(\alpha_{k,l}^i\right)_{i\geq 2}$ et $\left(\beta_{k,l}^i\right)_{i\geq 2}$ (voir figure B.1).

On évalue alors chacun de ces termes en $x = x_0$. Remarquons tout d'abord que $\forall i, j\ \ k_i^j(x_0) = \frac{d^j((x-x_0)k_i)}{dx^j}(x_0) = j\frac{d^{j-1}k_i}{dx^{j-1}}(x_0)$ (d'après l'équation (B.2)) et que, si $x = x_0$, on a $x^* = x_0$ et $y^* = y_0$ de telle sorte que toutes les dérivées sont évaluées en (y_0, x_0). On a alors :

$$
\begin{aligned}
\frac{d^2\tilde{y}}{dx^2}(x_0) &= \frac{2}{6}\left(\frac{dk_1}{dx}(x_0)+2\frac{dk_2}{dx}(x_0)+2\frac{dk_3}{dx}(x_0)+\frac{dk_4}{dx}(x_0)\right)\\
&= \frac{1}{3}\left(\frac{\partial f}{\partial x}+k_1(x_0)\frac{\partial f}{\partial y}+\frac{\partial f}{\partial x}+k_2^1(x_0)\frac{\partial f}{\partial y}+\frac{\partial f}{\partial x}+k_3^1(x_0)\frac{\partial f}{\partial y}\right)\\
&= \frac{\partial f}{\partial x}+\frac{1}{3}.\frac{\partial f}{\partial y}\left(f(y_0,x_0)+k_2(x_0)+k_3(x_0)\right)=\frac{\partial f}{\partial x}+f(y_0,x_0)\frac{\partial f}{\partial y}\\
&= \frac{d^2y}{dx^2}(x_0)
\end{aligned}
$$

$$
\begin{aligned}
\frac{d^3\tilde{y}}{dx^3}(x_0) &= \frac{3}{6}\left(\frac{d^2k_1}{dx^2}(x_0)+2\frac{d^2k_2}{dx^2}(x_0)+2\frac{d^2k_3}{dx^2}(x_0)+\frac{d^2k_4}{dx^2}(x_0)\right)\\
2.\frac{d^3\tilde{y}}{dx^3}(x_0) &= \frac{1}{2}\left(\frac{\partial^2 f}{\partial x^2}+2k_1(x_0)\frac{\partial^2 f}{\partial x\partial y}+k_1(x_0)^2.\frac{\partial^2 f}{\partial y^2}\right)+\sum_{k=1}^{2}\sum_{l=1}^{k}\left(2\alpha_{k,l}^2+\beta_{k,l}^2\right)\frac{\partial^k f}{\partial x^{k-l}\partial y^l}\\
&\quad = 2\frac{\partial^2 f}{\partial x^2}+\frac{\partial^2 f}{\partial x\partial y}\left(k_1(x_0)+k_2^1(x_0)+2k_3^1(x_0)\right)+\frac{\partial^2 f}{\partial y^2}\left(\frac{k_1(x_0)^2}{2}+\frac{\left(k_2^1(x_0)\right)^2}{2}+\left(k_3^1(x_0)\right)^2\right)\\
&\quad +\left(k_2^2(x_0)+k_3^2(x_0)\right)\frac{\partial f}{\partial y}\\
&= 2\frac{\partial^2 f}{\partial x^2}+\frac{\partial^2 f}{\partial x\partial y}\left(f(y_0,x_0)+k_2(x_0)+2k_3(x_0)\right)+\frac{1}{2}.\frac{\partial^2 f}{\partial y^2}\left(f(y_0,x_0)^2+(k_2(x_0))^2+2\left(k_3(x_0)\right)^2\right)\\
&\quad +2\left(\frac{dk_2}{dx}(x_0)+\frac{dk_3}{dx}(x_0)\right)\frac{\partial f}{\partial y}\\
&= 2\frac{\partial^2 f}{\partial x^2}+4.f(x_0,y_0).\frac{\partial^2 f}{\partial x\partial y}+2.f(y_0,x_0)^2.\frac{\partial^2 f}{\partial y^2}+\left(2\frac{\partial f}{\partial x}+\frac{\partial f}{\partial y}.\left(k_2^1(x_0)+k_1(x_0)\right)\right)\frac{\partial f}{\partial y}\\
&= 2\frac{d^3y}{dx^3}(x_0)
\end{aligned}
$$

$$
\begin{aligned}
\frac{d^4\tilde{y}}{dx^4}(x_0) &= \frac{4}{6}\left(\frac{d^3k_1}{dx^3}(x_0)+2\frac{d^3k_2}{dx^3}(x_0)+2\frac{d^3k_3}{dx^3}(x_0)+\frac{d^3k_4}{dx^3}(x_0)\right)\\
\frac{3}{2}\frac{d^4\tilde{y}}{dx^4}(x_0) &= \frac{1}{4}.\left(\frac{\partial^3 f}{\partial x^3}+3.k_1(x_0).\frac{\partial^3 f}{\partial x^2\partial y}+3.k_1(x_0)^2.\frac{\partial^3 f}{\partial x\partial^2 y}+k_1(x_0)^3.\frac{\partial^3 f}{\partial y^3}\right)\\
&\quad +\frac{1}{4}.\left(\frac{\partial^3 f}{\partial x^3}+3.k_2^1(x_0).\frac{\partial^3 f}{\partial x^2\partial y}+3.\left(k_2^1(x_0)\right)^2\frac{\partial^3 f}{\partial x\partial y^2}+\left(k_2^1(x_0)\right)^3\frac{\partial^3 f}{\partial y^3}\right)+2\alpha_{1,1}^3.\frac{\partial f}{\partial y}+2\alpha_{2,1}^3.\frac{\partial^2 f}{\partial x\partial y}\\
&\quad +2\alpha_{2,2}^3.\frac{\partial^2 f}{\partial y^2}+\frac{\partial^3 f}{\partial x^3}+3k_3^1(x_0)\frac{\partial^3 f}{\partial x^2\partial y}+3k_3^1(x_0)^2\frac{\partial^3 f}{\partial x\partial y^2}+k_3^1(x_0)^4\frac{\partial^3 f}{\partial y^3}+\beta_{1,1}^3.\frac{\partial f}{\partial y}+\beta_{2,1}^3.\frac{\partial^2 f}{\partial x\partial y}+\beta_{2,2}^3.\frac{\partial^2 f}{\partial y^2}\\
&= \frac{3}{2}\left(\frac{\partial^3 f}{\partial x^3}+3f(y_0,x_0).\frac{\partial^3 f}{\partial x^2\partial y}+3.f(y_0,x_0)^2.\frac{\partial^3 f}{\partial x\partial y^2}+f(y_0,x_0)^3\frac{\partial^3 f}{\partial y^3}\right)\\
&\quad +\frac{\partial f}{\partial y}\left(k_2^3(x_0)+k_3^3(x_0)\right)+\frac{\partial f^2}{\partial y^2}\left(\frac{3}{2}k_2^1(x_0)k_2^2(x_0)+3k_3^1(x_0)k_2^2(x_0)\right)+\frac{\partial^2 f}{\partial x\partial y}\left(\frac{3}{2}k_2^2(x_0)+3k_3^2(x_0)\right)\\
&= \frac{3}{2}\left(\frac{\partial^3 f}{\partial x^3}+3f(y_0,x_0).\frac{\partial^3 f}{\partial x^2\partial y}+3.f(y_0,x_0)^2.\frac{\partial^3 f}{\partial x\partial y^2}+f(y_0,x_0)^3\frac{\partial^3 f}{\partial y^3}\right)+\frac{\partial f}{\partial y}\left(3\frac{d^2k_2}{dx^2}(x_0)+3\frac{d^2k_3}{dx^2}(x_0)\right)\\
&\quad +\frac{\partial^2 f}{\partial x\partial y}\left(3\frac{dk_2}{dx}(x_0)+6\frac{dk_3}{dx}(x_0)\right)+\frac{\partial^2 f}{\partial y^2}\left(3f(y_0,x_0)\frac{dk_2}{dx}(x_0)+6f(y_0,x_0)\frac{dk_3}{dx}(x_0)\right)\\
\frac{d^4\tilde{y}}{dx^4}(x_0) &= \frac{\partial^3 f}{\partial x^3}+3f'(y_0,x_0)\frac{\partial^3 f}{\partial x^2\partial y}+3f'(y_0,x_0)^2\frac{\partial^3 f}{\partial x\partial y^2}+f(y_0,x_0)^3\frac{\partial^3 f}{\partial y^3}\\
&\quad +f(y_0,x_0)\left(2\frac{dk_2}{dx}(x_0)+4\frac{dk_3}{dx}(x_0)\right)\frac{\partial^2 f}{\partial y^2}+\frac{\partial^2 f}{\partial x\partial y}\left(2\frac{dk_2}{dx}(x_0)+4\frac{dk_3}{dx}(x_0)\right)+2\frac{\partial f}{\partial y}\left(\frac{d^2k_2}{dx^2}(x_0)+\frac{d^2k_3}{dx^2}(x_0)\right)\\
&= \frac{\partial^3 f}{\partial x^3}+3f'(y_0,x_0)\frac{\partial^3 f}{\partial x^2\partial y}+3f'(y_0,x_0)^2\frac{\partial^3 f}{\partial x\partial y^2}+f(y_0,x_0)^3\frac{\partial^3 f}{\partial y^3}+3f(y_0,x_0)\frac{df}{dx}(x_0)\frac{\partial^2 f}{\partial x^2}+\\
&\quad 3\frac{df}{dx}(x_0)\frac{\partial^2 f}{\partial x\partial y}+\left(\frac{\partial^2 f}{\partial x^2}+2f\frac{\partial^2 f}{\partial x\partial y}+f^2\frac{\partial^2 f}{\partial y^2}+\left(\frac{\partial f}{\partial x}+f\frac{\partial f}{\partial y}\right)\frac{\partial f}{\partial y}\right)\frac{\partial f}{\partial y}\\
&= \frac{\partial^4 y}{\partial x^4}(x_0)
\end{aligned}
$$

□

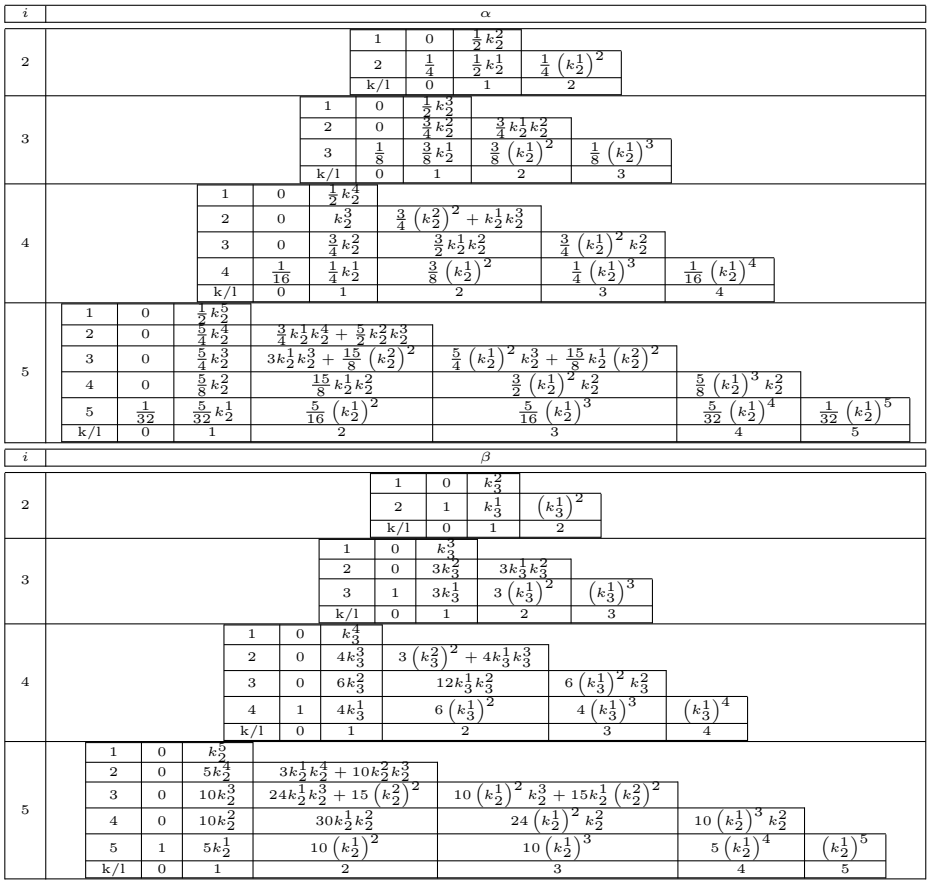


FIG. B.1 – Valeurs de $\alpha_{k,l}^i$ et $\beta_{k,l}^i$

Static analysis by abstraction interpretation of hybrid systems

Abstract The power and efficiency of abstract interpretation based static analysis methods for the verification of safety critical embedded software is no longer to be outlined. However, the precision of the analysers needs now to be reinforced. The use of ever more sophisticated relational abstract domains make it possible to reduce the overapproximation that limits the power of simpler domains, but the precision of the latest analysers is still constrained by the precision of the input values. These are given by a sensor that measures some physical phenomena, and they are generally overapproximated by an interval. In order to deal better with these continuous inputs, one may analyse, in addition to the program itself, the physical environment in which the program is executed. In this way, we obtain a more complex system which has both a discrete dynamics (the program) and a continuous one (the environment). Such hybrid systems are generally studied using extensions of finite automata or process calculi that introduce a continuous dynamics. The use of model-checking techniques still suffers from a combinatorial explosion that prohibits the use of such models for the analysis of large safety critical embedded programs.

The first contribution of this thesis is to provide an extension of imperative programming languages that describes both the program, the external environment and the interactions between both. The physical environment is described as a set of differential equations, each of them representing a continuous mode. The interactions between the program and the environment are modeled using two extra keywords that represent the effect of sensors and actuators. A denotational semantics is given to the whole system (the program and the environment together). This semantics remains very close to the standard denotational semantics of imperative languages. The main difficulty was to define a semantics for the continuous part of the system : we express the solutions of differential equations as the fixpoint of a monotone operator in a CPO, and we show that Kleene's iteration converges towards this fixpoint.

The second contribution is an abstract interpretation based static analysis of these hybrid systems. Our method has two stages. First, under some assumptions on the program to be analysed, a partitioning of the space of the input variables is built. For that, we combine a forward reachability method with an interval based analysis. In this way, we obtain an abstraction of the impact of the program on the continuous evolution : the continuous space is divided into zones in which we know that an actuator will be activated. The second stage of the analysis use this partitioning and a guaranteed integration of differential equations algorithm in order to build a safe overaproximation of the continuous evolution. A prototype analyser using these techniques was implemented and the first results on examples of hybrid systems from the literature show good results.

Finally, the third contribution of this thesis is a new guaranteed integration algorithm named GRKLib. Unlike existing methods, GRKLib is based on a non-validated, numerical integration scheme (we chose an order 5 Runge-Kutta method, but any other algorithm works). The global error made during the numerical integration is then computed using interval arithmetic. This error is expressed as the sum of three terms : the one-step error, the propagation error and the computation error due to the use of floating point numbers. Each term is computed separately and advanced techniques are used in order to reduce them and control the integration step size so that the growth of the global error is limited. A C++ library implementing these concepts was developed and the results presented in this thesis are very promising.

Keywords : *abstract interpretation ; hybrid systems ; denotational semantics ; ordinary differential equations ; guaranteed integration ; interval analysis.*

Analyse statique par interprétation abstraite de systèmes hybrides

Olivier Bouissou
Laboratoire Modélisation et Analyse de Systèmes en Interaction,
CEA - Centre de Saclay, F-91191 Gif-sur-Yvette Cedex

Résumé Si l'intérêt et l'efficacité des méthodes d'analyse statique par interprétation abstraite pour la vérification des programmes critiques embarqués ne sont plus à démontrer, il est maintenant nécessaire d'obtenir des méthodes les plus précises possibles. Si l'utilisation de domaines abstraits relationnels de plus en plus élaborés permet de diminuer la surapproximation dont souffre les domaines les plus simples, les analyses actuelles souffrent toujours d'une mauvais prise en compte des entrées du programme. Ces entrées sont fournies par un capteur qui mesure une grandeur physique, et sont généralement surapproximées par un intervalle. Une piste d'étude récente pour mieux gérer ces entrées continues consiste à étudier, outre le programme lui-même, l'environnement physique dans lequel il est exécuté. On obtient ainsi un système plus complexe comprenant une dynamique discrète (le programme) et une dynamique continue (l'environnement). L'étude de tels systèmes hybrides repose actuellement essentiellement sur des extensions des automates à états finis et des algèbres de processus introduisant une dynamique continue. L'analyse de ces systèmes par des techniques de *model-checking* souffre encore d'une explosion combinatoire excluant leur utilisation pour les logiciels embarqués critiques les plus gros.

La première contribution de cette thèse est une extension des langages de programmation impératifs permettant de décrire à la fois le programme, l'environnement extérieur et les interactions entre le programme et l'environnement. L'environnement physique est décrit par un ensemble d'équations différentielles représentant chacune un mode continu, et les interactions entre le programme et l'extérieur sont modélisés par deux mots clés représentant les capteurs et actionneurs. Nous donnons à l'ensemble (programme plus environnement physique) une sémantique dénotationnelle qui reste très proche de celle définie pour les langages impératifs classiques. La difficulté majeure dans la construction de cette sémantique a été de définir une sémantique pour la partie continue : les solutions des équations différentielles sont exprimées comme le plus petit point fixe d'un opérateur monotone dans un CPO, et nous montrons que les itérées de Kleene convergent vers ce point fixe.

La seconde contribution est une méthode d'analyse statique par interprétation abstraite de ces systèmes hybrides. Cette méthode fonctionne en deux temps. Tout d'abord, sous certaines restrictions portant sur le programme à analyser, on construit un recouvrement de l'espace des variables d'entrée via une analyse par intervalle couplée à une analyse d'atteignabilité en avant. On obtient ainsi une abstraction de l'impact qu'à le programme sur l'évolution continue : l'espace d'entrée du programme est découpé en zones dans lesquelles on est sûr qu'un actionneur sera activé. Dans un deuxième temps, nous utilisons ce recouvrement et une méthode d'intégration garantie des équations différentielles pour obtenir une surapproximation de l'évolution continue. Un analyseur prototype implémentant ces techniques a été développé et les tests sur les exemples classiques de systèmes hybrides montrent de bons résultats.

Enfin, la troisième contribution de cette thèse est une nouvelle méthode d'intégration garantie nommée GRKLib. Contrairement aux méthodes existantes, GRKLib se fonde sur un schéma d'intégration numérique non garantie (nous avons choisi un schéma de Runge-Kutta d'ordre 4, mais n'importe quelle autre convient) et nous calculons, en utilisant l'arithmétique d'intervalles, l'erreur globale commise lors de l'intégration numérique. Cette erreur s'exprime comme la somme de trois termes : l'erreur sur un pas, la propagation de l'erreur et l'erreur due aux nombres flottants. Chaque terme est calculé séparément et des techniques avancées permettent de les réduire et de contrôler au mieux le pas d'intégration pour limiter l'accroissement de l'erreur globale. Une librairie C++ implémentant ces concepts a été développée, et les résultats présentés dans cette thèse sont prometteurs.

Mots clés : *interprétation abstraite ; systèmes hybrides ; sémantique dénotationnelle ; équations différentielles ; intégration garantie ; analyse par intervalles.*